POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOTTORATO DI RICERCA IN INGEGNERIA DELL'INFORMAZIONE

System Support for Adaptive Performance and Thermal Management of Chip-Multiprocessors

Doctoral Dissertation of:
Filippo Sironi

Advisor:
Prof. Marco D. Santambrogio

Tutor:
Prof. Donatella Sciuto

Supervisor of the Doctoral Program:
Prof. Carlo E. Fiorini

2013 – XXVI

to my family and my love, since
"The family is one of nature's masterpieces."
(G. Santayana)

ABSTRACT

Computer architecture crossed a critical juncture at the beginning of the last decade. Single-thread performance stopped scaling due to technology limitations and complexity constraints. Therefore, chip manufacturers started relying on multi-threading and multicore processors to scale-up performance efficiently while keeping other figures of merit like energy and power consumption under control. In fact, whenever parallel software is available, a multicore processor harnessing Thread-Level Parallelism (TLP) can outperform a massive superscalar processor exploiting Instruction-Level Parallelism (ILP) within the same power budget. As a consequence, on-chip parallel architectures, which once were rare, are now commodity across all domains, from embedded and mobile computing systems to large-scale installations. Nevertheless, achieving efficient performance accounting for energy and power consumption progressively became increasingly complex requiring significant innovation across the hardware/software execution stack, even for commodity solutions.

At a high level, two challenges arise that hinder multicore processors efficiency. First, it must be possible to effectively partition hardware resources among co-runner applications within multi-program workloads and avoid the negative effects of sharing when hardware resource cannot be partitioned. Hardware resource partitioning is necessary because most multi-threaded applications do not fully exploit the parallelism available in commodity multicore processors due to the major difficulties of fine-grain parallelism. Among the hardware resources worth partitioning, there are: compute bandwidth and cores, and possibly others depending on the workload. Ideally, the system software layer of the hardware/software execution stack should act on hardware resource partitioning to attain fair application performance and provide Quality of Ser-

vice (QoS) guarantees while respecting system constraints. Second, the system software layer should operate in a transparent fashion without burdening application programmers with all the complexities of the hardware/software execution stack.

The focus of this dissertation is twofold. First, support efficient hardware resource partitioning for commodity multicore processors through a system software layer, which operate transparently for applications. To this end, I present solutions to attain fair application performance and provide QoS guarantees for co-runner applications within a multi-program workload accounting for application-specific performance measurements and performance goals. Second, support efficient Dynamic Thermal Management (DTM) for commodity multicore processors through a low-level system software layer. For this purpose, I present a solution to constrain temperature when a multi-program workload of single-threaded applications runs on a Chip-Multiprocessor (CMP). The resulting artifact is a set of changes, runtimes, and libraries for the GNU's not Unix (GNU)/Linux operating system.

On the performance side, I present the Heart Rate Monitor (HRM), Metronome, and Metronome++. First, HRM is a split-design subsystem consisting of an extension of the Linux kernel and a user-space library to attach applications to the subsystem. HRM addresses the impedance-mismatch problem by providing application-specific performance measurements that are meaningful to both programmers and users and, at the same time, useful to the system software layer of the hardware/software execution stack. libhrm provides programmers a simple Application Programming Interface (API) to instrument applications so as to define performance measurements and allow users to specify performance goals. HRM and libhrm make the operations of the system software layer I developed transparent to application programmers, which just exploit their knowledge of the application domain to define meaningful performance measurements. Second, Metronome is a kernel-space runtime introducing the notion of performance-aware fair scheduling by extending one of the scheduling classes of the Linux kernel. Metronome exploits HRM and the performance measurements it provides to drive application performance

towards performance goals for co-runner applications within a multi-program workload. Metronome achieves its goal by implementing simple compute bandwidth partitioning mechanism and policy. Third, Metronome++ is a leap ahead with respect to Metronome; it adopts a split-design across the kernel- and user-space. A user-space runtime drives the kernel-space extension of the scheduling infrastructure of the Linux kernel to provide QoS guarantees for co-runner applications within a multi-program workload by harnessing application characteristics like speedup and execution phases. Metronome++ achieves its goal by implementing compute core partitioning mechanism and policy. This dissertation additionally presents a set of minor achievements harnessing different decision-making techniques other than the heuristics Metronome and Metronome++ make use of.

On the temperature side, I present ThermOS, an extension of the Linux kernel providing DTM through formal feedback control and idle cycle injection. ThermOS addresses a shortcoming of commodity CMPs, which do not allow different cores to run at different clock frequencies when they operate in the same state. ThermOS avoids the negative effects depending on the lack of fine-grain control over hardware facilities like Dynamic Voltage and Frequency Scaling (DVFS) and improves upon state of the art.

On the performance/temperature side, I present preliminary results regarding joint adaptive performance and thermal management combining some of the aforementioned approaches.

SOMMARIO

L'architettura dei calcolatori ha attraversato un momento critico all'inizio dello scorso decennio. Le prestazioni dei processori dotati di un singolo core e capaci di eseguire un singolo thread[1] hanno smesso di aumentare in seguito a limitazioni tecnologiche e problemi di complessità. Di conseguenza, i produttori di processori hanno iniziato ad appoggiarsi all'esecuzione contemporanea di più thread tramite processori dotati di più di un core e capaci di eseguire più di un thread contemporaneamente;[2] questo ha permesso di aumentare le prestazioni di processori mantenendo livello di efficienza accettabili in termini di consumo energetico e dissipazione di potenza. In fatti, quando del codice parallelo è disponibile, un processore multicore che sfrutta il parallelismo a livello di thread può raggiungere prestazioni svariati ordini di grandezza più elevate rispetto ad un processore singlecore superscalare che sfrutta il parallelismo a livello di istruzione, questo raggiungendo lo stesso livello di dissipazione di potenza. In conseguenza, i processori multicore, che fino a poco tempo prima erano una rarità, si sono diffusi fino a diventare l'unica soluzione disponibile per tutti i calcolatori, dai dispositivi dedicati come gli smartphone e i tablet fino alle grandi installazioni. Nonostante l'introduzione dei processori multicore, raggiungere livelli di prestazioni ed efficienza elevati è diventato sempre più complesso; questo ha spinto i ricercatori ad introdurre continue novità a tutti i livelli dell'infrastruttura hardware/software.

I problemi che affliggono i processori multicore e che impediscono di raggiungere livelli di prestazioni ed efficienza elevati sono principalmente due. Primo, i processori multicore dovrebbero permettere di suddividere le loro

1 Nel prosieguo di questo sommario ci riferiremo ai processori dotati di un singolo core e capaci di eseguire un single thread come processori singlecore.
2 Nel prosieguo di questo sommario ci riferiremo ai processori dotati di più di un core e capaci di eseguire più di un thread contemporaneamente come processori multicore.

risorse tra le applicazioni che girano contemporaneamente evitando gli effetti negativi dovuti alla condivisione quando queste risorse non sono partizionabili. Il partizionamento delle risorse si è reso necessario con l'aumentare del numero di core perché la maggior parte del codice parallelo non è in grado di utilizzare al meglio i processori multicore odierni a causa delle difficoltà causate dal parallelismo a grana fine. Le risorse che sono più importanti da partizionare sono: il tempo di CPU e i core, e possibilmente altre come la cache, la banda verso la memoria, ecc., dipendentemente dall'insieme di applicazioni che sono in esecuzione. Idealmente, il software di sistema dell'infrastruttura hardware/software dovrebbe prendersi in carico del partizionamento delle risorse mantenendo un equità tra le applicazioni in esecuzione rispettando i livelli di qualità del servizio richiesto e i vincoli di sistema come il consumo energetico o la dissipazione di potenza. Il secondo problema che affligge i processori multicore e che impedisce di raggiungere livelli di prestazioni ed efficienza elevati riguarda il metodo con cui il software di sistema opera. Il software di sistema dovrebbe essere completamente trasparente senza tediare i programmatori di applicazioni esponendogli le complessità dell'infrastruttura hardware/software, tra le quali spicca il partizionamento delle risorse.

Gli obiettivi di questa tesi sono principalmente due. Primo, supportare il partizionamento efficiente delle risorse dei processori multicore attraverso un'estensione del software di sistema, che dovrà operare in modo trasparente. A questo scopo, presento svariate soluzioni per ottenere equità tra le applicazioni in esecuzione e raggiungere i requisiti di qualità del servizio quando più applicazioni sono contemporaneamente in esecuzione sullo stesso processore multicore. In aggiunta, l'estensione del software di sistema proposta in questa tesi supporta in modo efficiente il controllo di temperature dinamico. A questo proposito, presento una soluzione per mantenere la temperatura sotto controllo quando più applicazioni che sono in grado di sfruttare un singolo core vengono eseguite contemporaneamente su un processore multicore. Le estensioni del software di sistema proposte in questa tesi sono una serie di modifiche al kernel del sistema operativo GNU/Linux, una un insieme di librerie e di runtime.

Per quanto riguarda le prestazioni, il primo contributo di questa tesi è l'Heart Rate Monitor (HRM) che sta alla base degli altri contributi: Metronome e Metronome++. HRM è un'infrastruttura di monitoraggio delle prestazioni implementata in parte all'interno del kernel Linux ed in parte attraverso una libreria user-space che permette alle applicazioni di comunicare con il nuovo sottosistema che è stato introdotto all'interno del kernel. Tale libreria, che ho chiamato libhrm mette a disposizione dei programmatori di applicazioni una semplice interfaccia per instrumentare le applicazioni in modo tale che queste segnalino al sottosistema all'interno del kernel Linux quando hanno raggiunto un punto di esecuzione "importante". La frequenza con cui questi segnali vengono inviati permette di calcolare una misura delle prestazioni dell'applicazione che può essere poi utilizzata dagli utenti per definire degli obiettivi a livello di prestazioni, una sorta di qualità del servizio. Fatta eccezione per l'instrumentazione, che richiede l'intervento da parte dei programmatori per fare in modo che l'unità di misura delle prestazioni sia di interesse e sia comprensibile per l'utente, tutte le operazioni eseguite da HRM e libhrm sono del tutto trasparenti alle applicazioni. Metronome, il secondo contributo lato prestazioni di questa tesi, è un'estensione del Completely Fair Scheduler (CFS) dell'infrastruttura di scheduling del kernel Linux. Metronome introduce la nozione di "performance-aware fair scheduling"; il concetto di equità (temporale) alla base della classe di scheduling del kernel Linux, CFS, viene modificato per tenere in considerazione le prestazione e gli obiettivi prestazioni, che vengono misurate e fornite da HRM. Metronome raggiunge il suo obiettivo tramite un'euristica che guida il partizionamento del tempo di CPU tra le applicazioni che sono in esecuzione contemporaneamente. Il terzo contributo in ambito prestazioni di questa tesi è Metronome++. Metronome++ è un'evoluzione di Metronome che adotta un'architettura distribuita tra user- e kernel-space come quella di HRM. Il runtime user-space guida l'estensione dell'infrastruttura di scheduling del kernel Linux che è ovviamente implementata in kernel-space per fornire garanzie sul raggiungimento di predeterminati livelli della qualità del servizio per applicazioni che vengono eseguite contemporaneamente sullo stesso processore multicore. Metronome++ raggiunge il suo obiettivo attraverso un'euristica basata su un modello di speedup delle applicazioni e che sfrutta anche le fasi che le applicazioni attraversano durante la loro esecuzione. L'euristica pilota l'estensione dell'infrastruttura di scheduling del kernel Linux per assegnare più o meno core alle applicazioni. Questa tesi presenta anche una serie di piccole estensioni a HRM e Metronome++ che ne modificano le architetture e i metodi di decisione.

Lato temperatura, il contributo di questa tesi è ThermOS, un'altra estensione del kernel Linux che fornisce capacità per controllare la temperatura di esecuzione dei singoli core di un processore multicore quando questi eseguono diverse applicazioni contemporaneamente. ThermOS cerca di far fronte ad alcune lacune dei processori multicore che non permettono di utilizzare in modo sufficientemente fine i sistemi di controllo della dissipazione di potenza come il cambiamento di voltaggio e frequenza dei core. ThermOS adotta una tecnica software: l'iniezione di cicli di idle che viene pilotata dal un controllore proporzionale-integrale. Evitando gli effetti negativi del controllo per processore che derivano dall'utilizzo del controllo di voltaggio e frequenza dei core, ThermOS migliora rapporto tra prestazioni e temperatura in alcune situazioni rispetto allo stato dell'arte.

Questa tesi presenta infine dei risultati preliminari sull'unione dei sistemi di controllo delle prestazioni e temperatura.

La restante parte di questo documento è organizzato come segue. Il capitolo 1 riporta alcune considerazioni di carattere generale sui recenti sviluppi delle architetture dei calcolatori e sui problemi che ne derivano. Il capitolo effettua anche una disamina dell'Autonomic Computing, il filone di ricerca che meglio si sposa con le idee alla base di questa tesi. Lo stesso capitolo termina elencando i contributi di questa tesi e come questi sono stati esposti alla comunità di ricerca in termini di pubblicazioni. Il capitolo 2 spiega le scelte che hanno portato alla progettazione e implementazione di HRM. Il capitolo riporta anche una serie di consigli su come utilizzare il sistema di monitoraggio delle prestazione (*i.e.,* guida all'instrumentazione) e i risultati sperimentali che sono stati ottenuti. Questa parte della tesi si chiude con una disamina dei lavori che più sono vicini ad HRM e con un breve sunto del capitolo. Utilizzando la medesima struttura del capitolo precedente, il capitolo 3 descrive Metronome e Metronome++, i risultati sperimentali che sono stati raccolti e i lavori che più sono correlati. Il capitolo 4 spiega la progettazione e implementazione di ThermOS, riporta i risultati sperimentali ottenuto con l'artefatto e un confronto diretto con lo stato dell'arte. Lo stesso capitolo chiude con una disamina dei lavori legati a ThermOS e con un breve sunto. Il capitolo 5 propone una prima congiunzione di due sistemi di controllo all'interno del sistema operativo FreeBSD: un sistema di controllo delle prestazioni simile a Metronome e un sistema di controllo della temperatura semplificato simile a ThermOS. L'unione dei due sistemi di controllo si realizza tramite un euristica che viene valutata al termine del capitolo. In fine, il capitolo 6 riporta alcune considerazioni sul lavoro di tesi nel suo insieme e una serie di possibili sviluppi futuri, alcuni dei quali presentano già dei risultati sperimentali preliminari. L'appendice A riporta uno dei contributi più recenti di questo lavoro di tesi: una versione di Metronome++ che fa uso delle teoria del controllo e di un sistema di gestione delle risorse che agisce interamente in user-space.

ACKNOWLEDGMENTS

And here we go with the acknowledgments. . . again! A questo giro non sapevo proprio come cominciare e quindi sono partito con una frase ad effetto! Sono passati 8 anni da quanto ho iniziato a frequentare il Politecnico di Milano (PoliMi) e in questi 8 anni ho scritto questa particolare sezione che accompagna ogni tesi per 3 volte. . . , è arrivato il momento di scrivere questa sezione per la quarta e ultima volta. Il problema è che comincio non avere più parole per ringraziare le persone che in un modo o nell'altro mi sono state vicine in questi anni. La cosa bella è che fino a questo momento queste persone sono aumentate di numero, anche se con alcune i rapporti sono andati un po scemando causa distanza e nuovi interessi.

Come da copione (voglio essere banale e scontato), il primo grazie è per i miei genitori: *Ornella* e *Renato*. Mamma e papà mi hanno insegnato che vivere non è sempre semplice, che le cose non vanno sempre come vorremmo e questo è stato utilissimo in questi 8 lunghi anni di Politecnico di Milano. . . perché le cose non sono sempre andate (subito) nel verso giusto. Grazie al vostro sacrificio economico e alla vostra sopportazione anche quando ero a qualche fuso di distanza, ho avuto la bellissima opportunità di frequentare un semestre all'estero. Vivere a Chicago, negli Stati Uniti per 5 mesi, frequentare un'università americana, è stata una fortuna immensa.

Un grazie speciale va anche alla mia sorellina. . . *Roberta* (Beba)! Sei probabilmente la persona che ha sempre creduto di più in me, tutte le volte in cui mi hai visto triste perché qualche cosa non andava nel verso giusto hai sempre trovato il modo di rinfrancarmi. Sei stata e sei tutt'ora un "benchmark" (cit. Renato) eccezionale!

Ora è arrivato il momento per un ringraziamento che non ho mai dovuto fare nelle 3 tesi che hanno preceduto questa. Un grazie di cuore alla mia

ragazza, *Marta*, per il suo amore durante questi ultimi 3 anni e per il supporto che non mi hai mai fatto mancare, indipendentemente da quale fosse la mia scelta... come quella di andarmene qualche mese a Boston, negli Stati Uniti. Anche nei momenti più bui (e ce ne sono stati) mi sei sempre rimasta a fianco e hai sempre creduto che insieme avremmo potuto farcela, persino quando io mi sono comportato come un "orso". C'è da dire che imparo... al secondo giro sono migliorato!

Un grazie tocca anche ai parenti più cari e ai quasi parenti che spero davvero lo diventi!

Un sentito grazie al mio relatore, *Marco D. Santambrogio*, per il suo supporto (morale e tecnico) e per il suo impegno. In questi anni il nostro rapporto è mutato, non siamo più solo professore e allievo ma anche amici. Grazie a te ho avuto modo di conoscere il programma di doppia laurea tra il Politecnico di Milano e la University of Illinois at Chicago (UIC) che, come ho già detto in precedenza, è stato di importanza fondamentale. Inoltre, hai spinto all'inverosimile per far si che andassi al Massachusetts Institute of Technology (MIT), non una, ma ben due volte. Un grazie va anche ad un altro membro del corpo docenti del PoliMi, *Donatella Sciuto*, che insieme a Marco mi ha permesso di frequentare il dottorato supportandomi tecnicamente ed economicamente.

Grazie a tutti gli amici e colleghi del NECST Lab (per me è e rimarrà per sempre MicroLab). Mi scuso in anticipo perché non riuscirò a ringraziarvi tutti; a mio discolpa, siamo veramente diventati tantissimi in questi ultimi 2 anni. Comincio con coloro che hanno contribuito allo sviluppo di questa tesi perché ovviamente non avrei potuto fare tutto da solo. Grazie a *Davide* (DBB), è stato con la tua tesi di laurea specialistica che abbiamo cominciato a "sporcarci" le mani con Linux (kernel) e con te ho portato avanti il primo tema di ricerca proposto in questa tesi. Ricordo ancora le notti insonni per consegnare gli articoli di PPoPP, DAC, ICAC, IPDPS, PACT e ancora DAC! Non siamo quasi mai riusciti a pianificare tutto per tempo... almeno c'è ancora spazio per migliorare. Grazie a *Riccardo* (Catta), con te abbiamo cominciato il lavoro sul secondo tema di ricerca proposto in questa tesi. Anche con te ci sono state delle belle giornate di lavoro con FreeBSD che non voleva proprio saperne di funzionare come volevamo noi. Ricorderò a lungo il fine settimana trascorso al PoliMi per la nostra prima volta a DAC. Un grazie di gruppo a *Simone* anche se dopo il 2012 non siamo più riusciti a lavorare seriamente insieme, *Martina* che è la mia controllista di fiducia (cibooo?!), *Gianluca* che non dice mai di no (anche quando dovrebbe), *Matteo* (Carteo) e *Jacopo*. Grazie anche ai più vecchi membri del lab, quelli che ancora ne fanno parte e quelli che ci hanno lasciato per cominciare a lavorare invece di continuare a giocare a bam-bam oppure a scavare delle fosse per cercare acqua! Non posso veramente citarvi tutti e mi dispiace.

It's now time to switch in English for a few sentences. I'd like to thanks my project and office mates at MIT for the work and the fun (food & drinks) we had together: *Charles*, *Chris*, *Harshad*, *Nathan*, and *George*. A few words also for my *M. Frans Kaashoek* and *Nickolai Zeldovich*; I didn't spend much time at MIT but I really learned a lot from the two of you, you really are amazing researchers. A final word of thanks for my dear *Deirdre*. Thanks you so much for the time I spent in your wonderful house with you a my big *Mr. Kitty*!

Tempo di tornare all'italiano... Grazie a tutti gli amici degli anni che hanno preceduto il dottorato. Anche qui, citarvi tutti è impossibile, siamo tantissimi e il fatto che ci si veda ancora almeno una o due volte l'anno (con alcuni anche più spesso) per una cena come si deve è eccezionale.

Grazie al mio storico gruppo di amici, è sempre un piacere rivedervi anche se ultimamente, con le trasferte negli Stati Uniti e nel resto del mondo e negli ultimi due anni in Valtellina, ci vediamo un po' di meno (non mi sono mai dimenticato di voi, capito *Luca*?!). C'è anche da dire che qualcuno non mi lascia mai tranquillo, neppure alle 7:15 sul treno... vero *Lorenzo*?!

Credo di aver finito, questa volta mi sono davvero dilungato e vi assicuro che dopo questi lunghissimi ringraziamenti (non sono mai stato così prolisso) ho un po' gli occhi lucidi.

Filippo

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

$(\text{PA})^2$	Performance-Aware Processor Allocator. 67
ACOS	Autonomic Operating System. 14, 54, 133
ADAM	Adaptation Manager. 16
API	Application Programming Interface. viii, 21, 22, 24, 25, 29–31, 33, 34, 36
ARMA	Auto-Regressive Moving Average. 139, 153, 154
AVC	Advanced Video Coding. 27, 141
CFS	Completely Fair Scheduler. xiii, 51, 54, 57–60, 63, 64
CGROUP	Control Group. 16
CMP	Chip-Multiprocessor. viii, ix, 14, 17, 19, 37, 38, 51–54, 73, 81, 84, 85, 88–92, 98, 109, 113, 114, 116–119, 123, 129, 130, 134
CPO	Continuous Program Optimization. 22
CPU	Central Processing Unit. 6, 13, 14, 16, 21–25, 51, 54–60, 63, 66–68, 70, 71, 73, 74, 76, 78, 79, 81, 84, 85, 90, 94, 96, 122–124, 128, 129, 133, 134
CRAC	Computer Room Air Conditioning. 89, 90
DPTM	Dynamic Performance and Thermal Management. 15, 18, 120–123, 126, 128–131, 133, 134
DRAM	Dynamic Random-Access Memory. 85
DTM	Dynamic Thermal Management. viii, ix, 14, 18, 86–91, 93–95, 98–100, 109–111, 116–122, 129

DTS Digital Thermal Sensors. 98, 99

DVFS Dynamic Voltage and Frequency Scaling. ix, 14–16, 53, 85, 89–91, 93, 107, 111–113, 118, 121, 122

EDF Earliest Deadline First. 57, 154

EWA Exponential Moving Average. 71, 125, 126

FIFO First In First Out. 57

GNU GNU's not Unix. viii, 21, 29, 133, 140, 144

HAL Hardware Abstraction Layer. 53

HRM Heart Rate Monitor. viii, xiii, 13, 14, 16, 17, 19, 21–41, 48, 49, 54, 59, 61–63, 65–67, 73–75, 85, 127, 140, 153, 154

HTT Hyper-Threading Technology. 118, 128, 144

HTTP HyperText Transfer Protocol. 26, 94

HTTPS HyperText Transfer Protocol over Secure Socket. 26

HVAC Heat, Ventilation, and Air Conditioning. 90

I/O Input/Output. 90, 91, 96, 134

IAAS Infrastructure-as-a-Service. 17

IBM International Business Machines. 6, 52

ILP Instruction-Level Parallelism. vii, 4, 53

IPC Instructions Per Cycle. 22, 24, 84, 85, 96, 138, 145

ISE Integral Squared Error. 76, 86

IT Information Technology. 20

KVM	Kernel Virtual Machine. 17
LLC	Last-Level Cache. 37, 38, 40, 49, 61, 66, 75, 107, 128, 134, 138, 144
MA	Moving Average. 71
MAPE	Mean Absolute Percentage Error. 145
MAPE-K	Monitor-Analyze-Plan-Execute with Knowledge. 11
MPEG	Moving Picture Experts Group. 27, 141
MPKI	misses per thousand instructions. 24
MPSOC	Multiprocessor System-on-Chip. 88
MSR	Model-Specific Register. 127
MTTF	Mean Time To Failure. 88, 120
ODA	Observe-Decide-Act. 11–13, 19, 51, 133
PAAS	Platform-as-a-Service. 17
PAAS	Performance-as-a-Service. 17
PAFS	Performance-Aware Fair Scheduler. 57
PAPI	Performance Application Programming Interface. 21
PEM	Performance and Environment Monitoring. 22
PLP	Process-Level Parallelism. 4, 33
PMU	Performance Monitoring Unit. 21–25
POSIX	Portable Operating System Interface for Unix. 24, 29, 140, 144
PROCFS	process information pseudo file system. 33
QOS	Quality of Service. vii, ix, 26, 55, 56, 119, 137–144, 146, 148, 154, 155

RLS Recursive Least Squares. 141, 142, 154

RR Round-Robin. 57

RSDL Rotating Staircase Deadline. 58

RWLOCK read/write spinlock. 33, 36

SLA Service-Level Agreement. 87, 119–126

SLO Service-Level Objective. 5, 6, 13–17, 19–21, 23, 24, 28, 30, 31, 33, 35, 43, 51, 54, 57, 59–63, 66, 67, 71, 74, 76, 78–85, 87, 119–131, 133, 134

SMT Simultaneous Multi-Threading. 116, 118

TCO Total Cost of Ownership. 120

TLB Translation Lookaside Buffer. 35

TLP Thread-Level Parallelism. vii, 4, 53

VLSI Very-Large-Scale Integration. 120

VM Virtual Machine. 4, 5, 53, 134, 135

XML eXtensible Markup Language. 21, 22

1 INTRODUCTION

The work we will describe in this dissertation best-fits the grand picture proposed by the autonomic computing movement, which is a fairly recent research area within the computer science community. Even though this dissertation finds itself at home within the autonomic computing movement, each of the chapters makes contributions we believe are valuable to different research areas within the computer science community: from computer architectures to operating systems.

We start with a broad overview of some of the hot topics in computer science that affect the design and implementations of modern computing systems listing the challenges researcher have to counter, with particular focus on those addressed in this dissertation. Since this dissertation leverages the "autonomic computing approach", we continue with brief introduction on this research line, which may not be common knowledge among the readers. Finally, we describe the contributions of this dissertation listing the published material that supports them.

1.1 GENERAL OVERVIEW

The evolution of our society towards the information society, the need to spread information across the Internet, to elaborate and access an ever increasing amount of information through Big Data analysis and visualization led electronic engineering and computer science to design and develop a wide variety of computing systems and change the paradigm developers were accustomed to. Therefore, computing systems, which range from smart phones to tablets and the large-scale installations of Google and other key players, gained a pervasive presence in all the aspects of our lives. This trend, together with the ever-growing demands for performance and efficiency determined, in the last decade, a turning point for the computer architecture community.

The traditional structure with a singlecore processor in charge of performing all the computations got beyond its limits as a consequence of the failure of Dennards' scaling law [1]. In order abide to Joy's law: the peak computer speed doubles each year [2], chip manufacturers leveraged the still standing Moore's law [3] and multicore processors became the prominent solution in the whole range of computing systems, from smart phone to servers. The advent of multicore processors caused a huge paradigm shift for both systems' and applications' developers and introducing a completely new class of problems.

The amount of Instruction-Level Parallelism (ILP) superscalar processors could extract from the stream of instructions after years of evolution (*e.g.,* deeper pipelines to increase the clock frequency, multiple issues per clock cycle, speculative and out-of-order execution to leverage the branches' history and the absence of true dependences) stopped scaling at historical rate with showing no sign of growth among successive families of processors (*e.g.,* Intel Haswell features negligible improvements from this standpoint with respect to Intel Ivy Bridge). Systems' and applications' developers are forced to leverage coarse-grain parallelism by means of Thread-Level Parallelism (TLP) as it happens for software meant to run on a single machines or Process-Level Parallelism (PLP) as it happens for software meant to exploit small- to large-scale installations. However, the complexity of exploiting multicore and manycore processors (the latter being yet another emerging paradigm) at both the system- and application-level [4] gave birth to phenomenon such as multi-program workloads [5] and, at its extreme, servers consolidation through Virtual Machines (VMs) on top of physical machines.

The epicenter of the computation is rapidly shifting from the peripherals, the terminals we use to access the Internet, which are most of the time energy constrained (*e.g.,* smart phones, tablets, laptops, etc.), to centralized small- and large-scale installations. In addition to this trend, small to medium businesses saw concrete opportunities in sharing physical machines due to the increasing costs of running even small-scale installations giving birth to the clouds. Multi-program workloads and servers consolidation through VMs are the norm in these environments since they favor energy efficiency [6] by using less physical machines, a net advantage since physical machines are not (perfectly) energy proportional [7] yet.

As a result, in the near future we are going to deal with computing systems whose complexity may approach the limits of a few administrators because of the demands of multiple tenants (*e.g.,* achieve Service-Level Objectives (SLOs), etc.) and the physical constraints of the installations (*e.g.,* optimize performance under a power cap, etc.).

1.2 challenges and optimization problems

At a high level, two challenges arise that hinder multicore processors usability given their architecture and the non-functional requirements (*e.g.,* performance of applications, temperature of processors, etc.) of modern computing systems.

The first challenge pertains the automatic partitioning of hardware resources among co-runner applications within multi-program workloads according to user-specified SLOs and accounting for possible computing system requirements. However, there exists hardware resources for which partitioning cannot be implemented efficiently (*e.g.,* caches, memory bandwidth, etc.); in such a situation, automatically discarding the negative effects of sharing of hardware resources among co-runner applications within multi-program workloads becomes extremely important.

System software already supports partitioning for specific hardware resources. For example, hypervisors and operating systems provide facilities (*e.g.,* nice, quotas, etc.) to partition both compute bandwidth and cores among VMs and processes. Unfortunately, none of these facilities supports automatic partitioning of hardware resource accounting for user-specified SLOs.

The first problem we address in this dissertation is to automatize the partitioning of compute bandwidth and cores trying to achieve user-specified SLOs, by discarding the negative effects of sharing some of the hardware resources [8–10]. The second optimization problem we address assumes partitioning is in place and a fine-grained allocation is needed to respect system requirements (*i.e.,* temperature of processors) [11]. In addition, this dissertation touches the more complex optimization problem of providing users with SLOs guarantees for a subset of applications within a multi-program workload, while accounting for the temperature of processors [12, 9].

This dissertation tackles a second challenge, which is about the transparency of operations. We believe the automatic partitioning of hardware resources following user-specified SLOs and computing system requirements should be as transparent as possible to applications developers. This way, automatic partitioning of hardware improves multicore processors usability, while transparency ease their adoption.

1.3 AUTONOMIC COMPUTING OVERVIEW

A promising approach to address the problems exposed in the previous section is to move (or at least reduce) the burden of managing computing systems from administrators to computing systems themselves. This paradigm shift may happen if computing systems become able to self-manage their resources by autonomously making low-level decisions (*e.g.,* move a task from one Central Processing Unit (CPU) core to another one, change the clock frequency of a CPU, etc.) according to high-level objectives and constraints stated by users and administrators.

Such approach was proposed back in the 2000s by Paul Horn, from International Business Machines (IBM), under the name of autonomic computing. He published a manifesto outlining the predominant characteristics of autonomic computing systems, where the term “autonomic” was chosen referring to the autonomic nervous system in biological life, which is in charge of controlling the unconscious actions in a living body [13]. For instance, the autonomic nervous system monitors and controls the heart rate, the temperature, and a number of other vital parameters of a human body to ensure a steady state: the “homeostasis”. At the end of the last decade we assisted to a review of the autonomic computing idea, which was expressed in more pragmatic terms keeping what is good of the initial grand vision: the autonomous operations, the ability to reach a steady state, the use of high-level objectives and constraints [14], leaving behind the concept of applying only learning and evolutionary techniques to address the aforementioned problems.

The work we will describe later in this dissertation follows the most recent and pragmatic route of autonomic computing, even though we explored the application of machine learning in some side projects, which however is not the reported in the remainder of this document.

1.3.1 *Self-* Properties*

Computing systems embracing the autonomic computing paradigm should present a subset of what the autonomic computing community believes are common properties. The set of properties is the first step to reach self-adaptation [15] (*i.e.,* being able to autonomously adapt operations in face of changing conditions). We usually refer to these properties as self-* properties [13]; Salehie and Tahvildari [15] propose a taxonomy of these property, which we report in Figure 1. Even though the taxonomy was first devised for self-adaptive software, we believe it is compelling for all kinds of autonomic computing systems.

The figure lays down the self-* properties in a pyramid, showing a hierarchical organization divided into three levels:

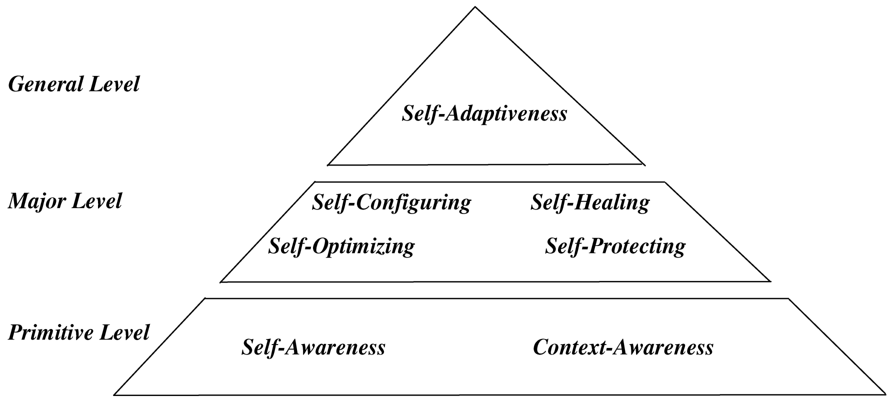


Figure 1.: Self-* properties taxonomy as proposed by Salehie and Tahvildari [15] (the figure is courtesy of Mazeiar Salehie and Ladan Tahvildari).

1. the primitive level defines the basic properties needed in order to support autonomous operations:
   - self-awareness, which means the computing system possesses knowledge of its internal state and behavior;
   - context-awareness, which refers to the knowledge of the environmental conditions in which the computing system operates;
2. the major level includes the properties initially identified within the autonomic computing manifesto:
   - self-configuration, the ability of the computing system to autonomously configure and reconfigure itself according to high-level policies;
   - self-optimization, the capability of the computing system to adjust the internal operations is carried on to yield the expected performance;[1]

1 In this case, performance is meant in a broader sense; the ability of performing as expected by matching objectives and respecting constraints.

- self-healing, the capacity of detecting, diagnosing, and repairing faults without affecting the internal operations (if feasible);
- self-protection, the potential of the computing system to defend against threats acting so as to mitigate or even avoid them;

these properties leverage both self- and context-awareness to understand the scenario the computing system finds itself in;

3. the general level including self-adaptiveness and self-organization where the first property refers to the computing system as a whole entity able to modify its behavior according to its internal state and environmental conditions while the second property emphasizes the computing system being formed by (semi-)independent subsystems able to orchestrate their work to achieve a given goal. Both self-adaptiveness and self-organization leverage the properties within the major level.

The proposed taxonomy comprises the largest possible set of self-* properties. Clearly, autonomic computing systems are not required to expose all of them when a subset may suffice to achieve their objectives and respect their constraints.

As an example, in this dissertation we will show two solutions to achieve adaptive performance management and one solution to achieve adaptive thermal management (see Chapter 3 and Chapter 4, respectively) with different approaches to awareness. Metronome and ThermOS support just self-awareness since the decision making processes are driven by local information only while Metronome++ also exploits context-awareness, since Metronome++ makes use of a richer set of data.

1.3.2 *Autonomic Control Loop*

In order to build computing systems that expose the aforementioned self-* properties we should adopt an alternative paradigm for their design and

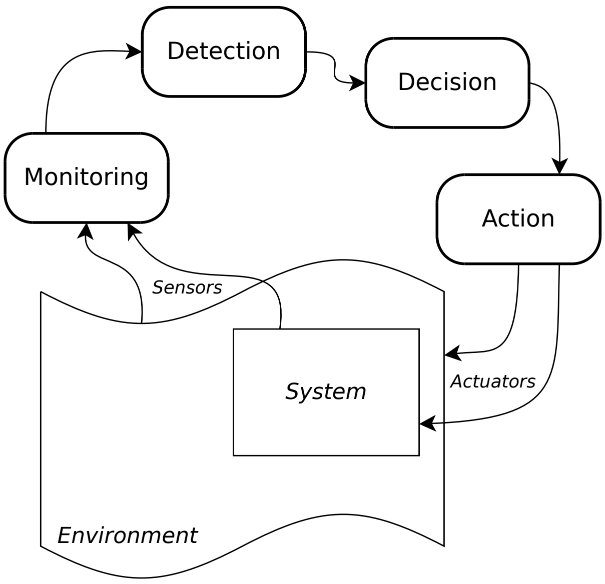


Figure 2.: Autonomic control loop (the figure is courtesy of Davide B. Bartolini).

implementation. This paradigm differs from the classic open loop solution and is characterized by a recurrent sequence of actions that consists in gathering information about the internal state and the environmental conditions (*i.e.,* achieving self- and context-awareness), detecting possible issues with respect to the objectives and constraints, eventually deciding a set of corrective actions to perform (*e.g.,* exploiting self-optimization, etc.), and then applying them. This paradigm is a classic close loop approach and a computing system designed and implemented around this solution is said to harness an autonomic control loop, thus becoming an autonomic computing system. Figure 2 graphically represents the autonomic control loop. This representation emphasizes the separation between the detection and decision phases. The detection process is in charge of analyzing the data coming from the sensors and to detect when something should be changed in order to restore the system from an unwanted state into its desired working conditions. The decision process is in charge

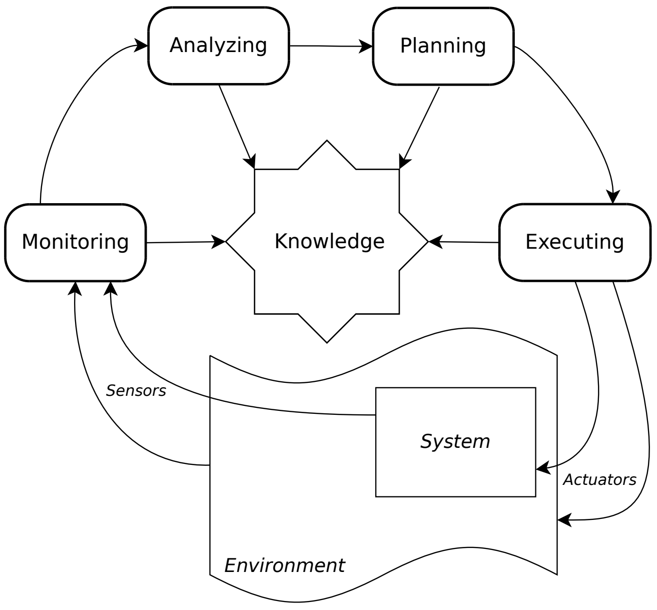


Figure 3.: Monitor-Analyze-Plan-Execute with Knowledge (MAPE-K) autonomic control loop (the figure is courtesy of Davide B. Bartolini).

of determining what should be changed (*i.e.*, picking the right action to be performed).

A second version of the autonomic control loop is the MAPE-K [16] and is graphically represented in Figure 3. This representation emphasizes the shared knowledge each phase of the autonomic control loop contributes to build. Moreover it chooses the verb "to plan" instead of "to decide", which poses more attention on the fact that decisions may affect the behavior of the computing system on the long run.

A third version of the autonomic control loop, which is also the one we favor within this dissertation, is called ODA [8] and it is graphically represented in Figure 4. In more details, the phases of ODA are concerned with the following operations:

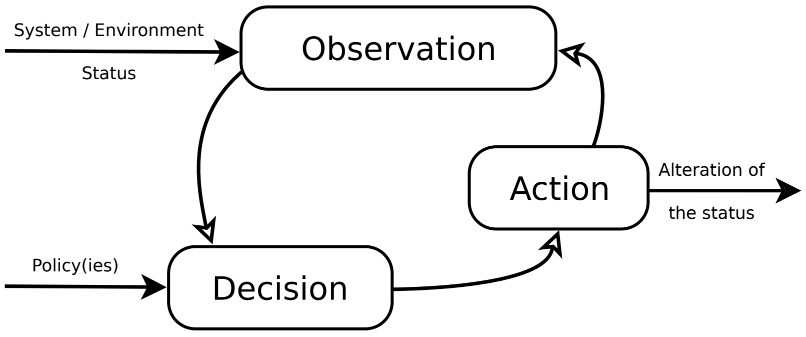


Figure 4.: Observe-Decide-Act (ODA) autonomic control loop (the figure is courtesy of Davide B. Bartolini).

- observation, collect information regarding both the internal state of the computing system and the environmental conditions it must face by means of monitoring infrastructures (*e.g.,* throughput or latency monitors, thermometers, etc.);

- decision, carry on the decision making process accounting for the data gathered through the observation phases, possibly considering the past iterations of the autonomic control loop, and come up with a set of actions to drive the current performance (in the broader sense of the term) towards the objectives while respecting constraints. The decision making process is usually carried on by adaptation policies or simply policies, which expect high-level objectives and constraints;

- action, perform the set/sequence of actions devised by adaptation policies through actuators, which are either physical or virtual devices that affect the internal operations of the computing system.

1.4 CONTRIBUTIONS

This section lists the contributions (in terms of the optimization problem they solve) spread throughout this dissertation pinpointing to the relevant publications.

- Filippo Sironi, Davide B. Bartolini, Simone Campanoni, Fabio Cancare, Henry Hoffmann, Donatella Sciuto, and Marco D. Santambrogio. Metronome: Operating System Level Performance Management via Self-Adaptive Computing. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 856–865, New York, NY, USA, 2012. ACM. 10.1145/2228360.2228514.

  The aforementioned paper is the first contribution of this dissertation and presents the design and implementation of early versions of Heart Rate Monitor (HRM) and Metronome. HRM is a performance monitoring infrastructure implementing the observation phase of ODA; its predominant characteristics are the: ease of use, generality, availability, meaningfulness, accuracy, and performance (*i.e.,* high scalability and low overhead). HRM provides high-level performance metrics and allows stating SLOs thanks to instrumentation. Metronome is an adaptation policy, more precisely a resource allocation policy plus a resource allocation mechanism, to distribute the CPU bandwidth among multi-thread applications within multi-program workloads. Metronome makes use of a heuristic decision making process and extends the default time-sharing scheduling class of the Linux kernel to drive applications towards user-stated SLOs. Metronome alongside with HRM solve the first optimization problem listed in Section 1.2: achieving user-specified SLOs for co-running applications.

- Davide B. Bartolini, Riccardo Cattaneo, Gianluca C. Durelli, Martina Maggio, Marco D. Santambrogio, and Filippo Sironi.[2] The Autonomic Operating System Research Project: Achievements and Future Directions.

2 Authors appear in alphabetical order.

In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 77:1–77:10, New York, NY, USA, 2013. ACM.
10.1145/2463209.2488828.

The paper reported above gives an high-level view of the Autonomic Operating System (AcOS) research project from which this dissertation rose; in fact, every one of the "full system" presented in this dissertation is a flavor of AcOS. In this paper, we present a novel contribution: Metronome++. Metronome++ is a sort of step forward with respect to Metronome; it is an adaptation policy featuring a model-based resource allocation policy and a resource allocation mechanism to move tasks among the Linux kernel runqueues. Metronome++ allows driving multi-thread applications within multi-program workloads towards user-stated SLO and it does so by exploiting applications characteristics like their speedup and the execution phases they traverse. Metronome++ alongside with HRM tackle again the first optimization problem listed in Section 1.2: achieving user-specified SLOs for co-running applications.

- Filippo Sironi, Martina Maggio, Riccardo Cattaneo, Giovanni Francesco Del Nero, Donatella Sciuto, and Marco Domenico Santambrogio. ThermOS: System Support for Dynamic Thermal Management of Chip Multiprocessors. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 41–50, Piscataway, NJ, USA, 2013. IEEE Press.
  10.1109/PACT.2013.6618802.

  The paper cited above describes the design and implementation of ThermOS. ThermOS couples a software solution to tackle the Dynamic Thermal Management (DTM) problem that is capable to overcome the limitations of Dynamic Voltage and Frequency Scaling (DVFS). In fact, DVFS in commodity multicore processors or Chip-Multiprocessor (CMP) has chip-wide side effects and may penalize the execution of applications from multiple tenants. Conversely, idle cycle injection acts on a per-CPU core basis and avoids this issue. Experimental results are encouraging since ThermOS achieves better performance than *Dimetrodon* [17] with the same temperature reduction and a superior trade-off than DVFS for small (*i.e.,* less than 30 % temperature reductions. ThermOS tackles the second optimization problem listed in Section 1.2: respecting computing system constraints expressed in terms of maximum operating temperature.

- Davide B. Bartolini, Filippo Sironi, Martina Maggio, Riccardo Cattaneo, Donatella Sciuto, and Marco D. Santambrogio. A Framework for Thermal and Performance Management. In *Proceedings of the Workshop on Managing Systems Automatically and Dynamically*, MAD '12, Berkeley, CA, USA, 2012. USENIX Association.

  The aforementioned paper describes an early design and implementation of ThermOS within the FreeBSD kernel instead of the Linux kernel coupled with a performance management solution similar to Metronome. The resulting system, which is called Dynamic Performance and Thermal Management (DPTM), has a better behavior than *Dimetrodon* and is capable of achieving user-stated SLOs while respecting administrator-stated temperature caps. This is the first work coupling adaptive performance and thermal management. DPTM tackles the combination of the first and second optimization problems listed in Section 1.2: achieving user-specified SLO for a subset of the co-running applications—as it happens for latency sensitive and batch workloads in Google datacenters [5]—and respecting computing system constraints expressed in terms of maximum operating temperature.

Apart from the main contributions of this dissertation, which are thoroughly described later in this document, there exist a number of small contributions and publications that are either incremental improvements or side projects originated from this dissertation.

- Filippo Sironi, Donatella Sciuto, and Marco D. Santambrogio. A Performance-Aware Quality of Service-Driven Scheduler for Multicore Processors. *SIGBED Rev.*, 2014.

This paper describes a complete user-space solution providing the functionality of HRM and Metronome++. We replicate the design of HRM within a user-space library without loosing in performance (sacrificing just availability since performance measurements and SLOs are not available in kernel-space). In addition, we exploit sound control theory to build the resource allocation policy in place of our model-based heuristic resource allocation policy. The implementation also leverages some of the functionality of the Linux kernel like Control Groups (cgroups) to aid both the performance monitoring infrastructure and the resource allocation mechanism, which exploits the cpuset subsystem of cgroups.

- Jacopo Panerati, Filippo Sironi, Matteo Carminati, Martina Maggio, Giovanni Beltrame, Piotr J. Gmytrasiewicz, Donatella Sciuto, and Marco D. Santambrogio. On Self-adaptive Resource Allocation through Reinforcement Learning. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems*, AHS '13, pages 23–30, Piscataway, NJ, USA, 2013. IEEE Press. 10.1109/AHS.2013.6604222.

- Jacopo Panerati, Martina Maggio, Matteo Carminati, Filippo Sironi, Marco Triverio, and Marco D. Santambrogio. *ACM Trans. Reconfigurable Technol. Syst.*, 2014.

  These papers explore the possibility of using reinforcement learning to learn resource allocation policies exploiting different actuators; in this work we employed both CPU core allocation and DVFS. HRM is the performance monitoring infrastructure of choice and, much like it happens with Metronome++, its consumer user-space capabilities aid the implementation of the "final system" which we call Adaptation Manager (AdaM). AdaM proved effective in learning adaptation policies for a considerable set of multi-thread applications.

We are already exploring a set of possible future work originating from this dissertation in the context of cloud computing. We report two of them whose abstracts were accepted for publication after a peer-review process.

- Davide B. Bartolini, Filippo Sironi, Martina Maggio, Gianluca C. Durelli, Donatella Sciuto, and Marco D. Santambrogio. Towards a Performance-as-a-Service Cloud. In *Proceedings of the 4th Symposium on Cloud Computing*, SoCC '13, New York, NY, USA, 2013. ACM. 10.1145/2523616.2525933

  This abstract describes the evolution of both Metronome and Metronome++ applied to the resource allocation problem in highly dynamic cloud computing infrastructure moving towards a Performance-as-a-Service (PaaS) cloud computing infrastructure. We believe this could be a compelling solution for both Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) solutions where user-stated SLOs are of great importance to both the users and the providers.

- Alberto Scolari, Filippo Sironi, Davide B. Bartolini, Donatella Sciuto, and Marco D. Santambrogio. Coloring the Cloud for Predictable Performance. In *Proceedings of the 4th Symposium on Cloud Computing*, SoCC '13, New York, NY, USA, 2013. ACM. 10.1145/2523616.2525955

  This abstract describes the design and implementation of Rainbow, a physical page allocator supporting page coloring [18] meant to be implemented inside hypervisors like the Kernel Virtual Machine (KVM) or Xen [19]. This is meant to be our first contribution towards the partitioning of the cache hierarchy in commodity CMPs and the first experimental results we collected with an early implementation within KVM are promising.

1.5 organization

The remainder of this dissertation is organized as follows. Chapter 2 introduces the problem of performance monitoring and describes the design and implementation of HRM alongside with an extensive experimental evaluation and a set of related work. Chapter 3 describes two of the contributions of this dissertation: Metronome and Metronome++. The chapter reports insights about the design and implementation of both coupled with extensive experimental results and an analysis of the relevant related work. Chapter 4 describes our finding in the context of DTM. It describes the design and implementation of ThermOS; the experimental results and a comparison wit the state of the art alongside with an analysis of the existing literature. Chapter 5 describes the design and implementation of DPTM, a framework that reunites adaptive performance and thermal management (with simplified implementations of Metronome and ThermOS within the FreeBSD operating system) with a heuristic approach. Finally, Chapter 6 concludes this dissertation with final remark and a brief discussion of possible future works. Appendix A described the design and implementation of a user-space implementation of Metronome++ exploiting control theory.

2 PERFORMANCE MONITORING

In the last decade computer architecture crossed a critical junction and so did the whole computing industry shifting from single to multicore processors. The introduction of Chip-Multiprocessors (CMPs) requires better support from system software: hypervisors, operating systems, and runtimes. System software must be able to manage emerging computer architectures achieving the same predictable performance single-thread applications used to have on singlecore processors. In addition, the growing importance of non-functional requirements (*e.g.*, performance, energy consumption, etc.) imposes system software to support the specification of Service-Level Objectives (SLOs) and the resource allocation mechanisms to meet them. Finding the proper resource allocations to yield the specified SLOs for multi-program workloads can be cumbersome; because of this, automatic resource allocation policies should close the loop by driving the available mechanisms. System software, operating systems in this specific case, are the perfect test bed for experimenting with various resource allocation policies and mechanisms as they find themselves in a favorable position in the hardware/software execution stack.

In this chapter we present the Heart Rate Monitor (HRM): a performance monitoring infrastructure for performance-aware computing to support the specification of high-level SLOs and the acquisition of performance measurements thanks to application instrumentation. HRM provides the functionality required to carry on the observation phase of the Observe-Decide-Act (ODA) autonomic control loop.

2.1 INTRODUCTION

The turn of computer architectures from the well understood, single processing element structure to multiple (possibly heterogeneous) processing elements is pervasive. This change has been dictated by physical (*i.e.,* inability to increase the clock frequency without incurring unmanageable power consumption) and architectural (*i.e.,* diminishing performance returns from efforts in further optimizing the performance of single processing elements) constraints [20]. To survive its commitment to exponential performance improvements [2], the computer industry changed its strategy, leading to the multicore processors era.

In the singlecore processors era, faster processors provided software performance improvements and applications experienced the so-called "free lunch", with free-of-charge speedups just by switching to the next-generation processor. The new parallel course in computer architectures, despite being due to computer architectural causes, carries the side effect of ending the "free lunch", posing a considerable burden of improving performance on both systems' and applications' developers. The demands for efficient and reliable parallel software sums up to the already considerable bulk of expertise software developers need to successfully cope with requirements for computing performance, functionality, reliability, and constraints satisfaction due to today's Information Technology (IT). Moreover, computational resources must be carefully managed to avoid hitting power and thermal limits, while respecting SLOs. This situation leads to an increased need of pushing as much of the computing system management as possible into computing systems themselves, making autonomic computing a possible breakthrough for IT success [13].

Respecting SLOs employing the least amount of resources is one of the objective of autonomic computing [13]. Autonomic computing systems are required to monitor their internal state and the environmental conditions, detect significant changes, decide a chain of actions, and actuate them [15]. The activity of gathering runtime information (referred to as either observation or monitor phase) is crucial, and the availability of accurate and appropriate status information can determine the efficacy of the computing system.

This chapter propose HRM, a performance monitoring infrastructure for power-aware computing to allow applications specifying SLOs and the operating system acquiring performance measurements. HRM is implemented inside the Linux kernel (even though an implementation for the FreeBSD kernel does exist [12]) and allows the kernel itself to access performance measurements, a net improvement compared to previous work [21].

The remainder of this chapter is organized as follows. Section 2.2 discusses related work and compare the main contribution of this chapter (*i.e.,* HRM) with the state of the art in performance monitoring infrastructures. Section 2.3 describes the design and implementation of HRM while Section 2.4 reports its extensive evaluation. Finally, Section 2.5 concludes the chapter.

2.2 RELATED WORK AND STATE OF THE ART

Traditionally, fine-grain performance measurements were collected through Performance Monitoring Units (PMUs) [22], which are sets of hardware registers in charge of counting hardware events like the number of unhalted cycles and the number of retired instructions. Different frameworks have been proposed to ease the use of PMUs. The Performance Application Programming Interface (PAPI) [23, 24] aims at creating a consistent interface towards PMUs of different Central Processing Units (CPUs). This Application Programming Interface (API) provides a unified interface, but it cannot deal completely with the heterogeneity of PMUs of different CPUs. A different approach is taken by Sprunt [25], who proposes the *Brink* and *Abyss* tools, which provide a high-level interface to the Intel Pentium 4 Processors PMU on GNU's not Unix (GNU)/Linux operating system. These tools rely on eXtensible Markup Language (XML) descriptions of the PMU capabilities, the desired configuration, and the applications to be monitored. However, both *brink* and *abyss* are tailored towards the Intel Pentium 4 Processors, which is quite heterogeneous family on its own, and further extensions are needed to support more CPUs. These APIs and tools allow to synthesize metrics based on the hardware events like the Instructions Per Cycle (IPC).

An alternative to PMUs, which proved themselves more profitable offline than online, is the use of software performance monitoring infrastructures. A notable example is the Performance and Environment Monitoring (PEM) [26] devised for supporting Continuous Program Optimization (CPO) an autonomic computing system, initially in the context of the K42 operating system [27]. PEM provides a repository of XML-specified events—at either computer architecture, operating system, or application level—and a set of tools to build "sensors" stubs from the XML specifications leveraging an instrumentation API. PEM has the merit of potentially support multiple programming language with ease thanks to the XML specifications. However, the high generality of PEM becomes a major drawback because of the burden it poses on systems' and applications' developers, who are in charge of providing both XML specifications and implement the "sensors" stubs.

The researchers behind the *Tessellation* operating system [28] acknowledged the need for a performance monitoring infrastructure supporting applications' developers and users understandable performance metric [29, 30]. A research project going into this direction is *Application Heartbeats* [21], which define the concepts for a performance monitoring infrastructure satisfying both applications' developers and users.

2.2.1 *Discussion*

When we first design and implemented HRM we asked ourselves whether existing performance monitoring infrastructures like the *Application Heartbeats* [21] reference implementation provided all the functionality we needed and, eventually, what they were missing. After a thorough analysis of the related work, we came up with the following set of requirements:

Table 1.: Comparison of performance monitoring alternatives

	PMUs	*Applications Heartbeats (ref.)*	HRM
ease of use	✓	✓	✓
generality		✓	✓
availability	✓		✓
meaningfulness		✓	✓
accuracy		✓	✓
performance			✓

- ease of use, the first and possibly most important characteristic of the performance monitoring infrastructure was to be easily usable by both systems' and applications' developers;
- generality, the performance monitoring infrastructure should be portable among different CPUs' families with little to no effort;
- availability, the performance monitoring infrastructure should provide its SLOs and performance measurements to the widest variety of actors within a computing system;
- meaningfulness, the performance metric must also be meaningful for applications' developers and especially for users, which may want to state a SLO;
- accuracy, the performance metric employed by the performance monitoring infrastructure must be representative since later on they are going to play an important role in resource allocation;
- performance, the performance monitoring infrastructure must scale properly on multicore processors when given more cores and should limit the overhead on applications.

Among the state of the art solution we analyzed traditional solutions such as the use of PMUs available in CPUs and the *Application Heartbeats* reference implementation.

Table 1 compares the aforementioned state of the art solutions with HRM. From both a systems' and an applications' developer standpoint PMUs and *Application Heartbeats* are equivalently accessible since it is possible to design a simple API that hides most of the idiosyncrasies of the underlying implementation. Though possible, designing a simple API for PMUs [23, 24, 31, 32] may be harder. However, the use of PMUs may be especially beneficial for applications' developers that can even avoid instrumenting their applications, thus providing the possibility to use a passive performance monitoring infrastructure instead of an active one such as *Application Heartbeats*.

Unfortunately, PMUs lack generality since they may require a lot of effort from the performance monitoring infrastructure provider to support many different CPUs [23, 24, 31]. Generality is most likely the best characteristic of *Application Heartbeats*, whose interface can be easily implemented by leveraging the Portable Operating System Interface for Unix (POSIX) API and nothing more as in the reference implementation [21].

PMUs provide high availability since the performance measurements they compute are accessible both in kernel- and user-space following almost the same procedure. *Application Heartbeats* could provide the same level of availability but, unfortunately, its reference implementation does not since it is limited to the user-space [21].

All of the performance metric collected through PMUs lack meaningfulness for both applications' developers and especially for users. We cannot require users to state SLOs in terms of IPC or last-level cache misses per thousand instructions (MPKI). Conversely, *Application Heartbeats* proposed a work-related performance metric that is simpler to under for both applications' developers, who are well-aware of their domain, and users, who can simply tell how many requests/s their web server should be able to serve.

Similar considerations hold for accuracy; past research showed that IPC is not a trustworthy performance metric for multi-thread applications [33] while work-related performance metrics are.

When it comes to performance both PMUs and the *Application Heartbeats* reference implementation fall short [8]. The former can incur overhead of up to 20 % while the latter forces serialization and executes a system call on the most used API call: `heartbeats()`, which may change applications' characteristics (*i.e.,* scalability) [4, 34] or compromise the state of functional units inside CPUs' cores [35, 36].

To avoid the limitations of both PMUs and the *Application Heartbeats* reference implementation we designed and implemented HRM, which we will discuss later in this section.

2.3 DESIGN AND IMPLEMENTATION

HRM borrows concepts defined by *Applications Heartbeats* to guarantee ease of use, generality, meaningfulness, and accuracy. However, HRM heavily changes the design and implementation of the performance monitoring infrastructure to improve the availability thanks to a kernel/user-space partitioned implementation and solve the performance issues by means of a multicore processors-awareness.

Let us start this section with few definitions to better understand the key insights behind the design and implementation of HRM.

2.3.1 *Definitions*

APPLICATION An application is any program in execution; it can be a single process, a set of processes, or a set of threads belonging to either the same process or not. Since applications are the entities of interests for a performance monitoring infrastructure, the performance monitoring infrastructure provider must take care of supporting any kind of application, independently of its structure. Because of this reason we define the concept of group.

GROUP A group is an ensemble of tasks that share the performance monitoring, where task is either a process or a thread following the Linux kernel nomenclature [37]. The concept of group is intended to allow the performance monitoring of applications that exploit parallelism in various ways; HRM improves upon the *Application Heartbeats* reference implementation since it supports applications harnessing process-level parallelism instead of applications exploiting just thread-level parallelism. This theoretically allows HRM to support performance monitoring for all of the configuration of the Apache HyperText Transfer Protocol (HTTP) Server [38].

Example of groups can be: an instance of the Apache HTTP Server handling `http://www.foo.com`, another instance handling `https://www.foo.com`, and another one handling `http://www.bar.com`. This allows to get statistics such as the number of requests/s served by the three instance of the Apache HTTP Server and in particular two different performance measurements for the HTTP and HyperText Transfer Protocol over Secure Socket (HTTPS) web site of the first domain, which may help providing different level of Quality of Service (QoS).

Following the concepts behind *Application Heartbeats*, we ask applications' developers themselves, which are well-aware of their domain, to signal the points of interest in their software through instrumentation. *Application Heartbeats* consider the "address space" the entity of interest for the performance monitoring while we introduced the concept of group of tasks, which is the entity of interest for the instrumentation. Since HRM require instrumentation, we strive to make this operation as simple as possible so as to retain the ease of use of *Application Heartbeats*. The most important part of the instrumentation consists in the addition of the call that emits heartbeats.

HEARTBEAT A heartbeat is a signal sent from a group (*i.e.,* either one of the tasks belonging to the group) to the performance monitoring infrastructure and means the group reached a points of interest.

The concept of heartbeat is extensively used in the literature to signal availability [39]. For example, each node within a high availability cluster may emit heartbeats with a constant pace to tell the master node that everything is proceedings; whenever the master node does not see heartbeats coming from a node it can perform corrective actions like restarting the computations that node was responsible for on other nodes as it happens in the *MapReduce* framework [40].

Within *Application Heartbeats* and HRM a heartbeat means one of the tasks within a group reached a point of interest in the execution that means the application is proceeding. For example, let us consider the *x264* video encoder [41], which implements the H.264/Moving Picture Experts Group (MPEG) 4 Part 10 or Advanced Video Coding (AVC) standard, included in the PARSEC 2.1 benchmark suite [42]. The parallel algorithm of *x264* harnesses a virtual pipeline with one stage per frame. *x264* processes in parallel a number of pipeline stages equal to the number of encoder threads realizing a sliding window moving from the beginning to the end of the pipeline. Once the execution of a stage finishes the encoder thread handling the stage issues a heartbeat, which in the context of *x264* signifies the completion of a frame.

The concept of heartbeat suits well the throughput-oriented applications where the concept of unit of work is well-defined. Usually, the section of code that takes care of a unit of work in on the critical path of an application and thus is relevant from a performance monitoring standpoint.

HOTSPOT A hotspot is a performance-critical section of code that gets executed by one or more tasks (*i.e.,* in serial or in parallel).

HRM requires that a group of tasks take care of executing a single hotspot. We understand this may be a limitation for those applications with multiple hotspots and with task pools in which tasks may execute any of the hotspots; because of this reason, we allow each task to connect to multiple group at the same time and specify on which hotspot it is active. This functionality was exploited in the context of a master thesis derived from this work [43].

```
for (int i = 0; i < n; ++i) {
        // encode the i-th frame
        ...
        heartbeat();
}
```

(a) Video encoder hotspot.

```
while ((fd = accept(lfd, ...) != -1) {
    if (fork() == 0) {
        // handle the fd connection
        ...
        heartbeat();
        exit(...);
    }
}
```

(b) Web server hotspot.

Figure 5.: Example hotspots.

Figures 5a and 5b show two sample sections of code representing hotspots in a single-thread video encoder and a multi-process web server, respectively.

Because of the throughout-oriented nature of the proposed performance monitoring infrastructure, each call to `heartbeat()` is placed on one of the critical paths of the code and thus must be lightweight to avoid the aforementioned issues. Because of this reason, HRM is designed and implemented to favor this call rather than any other call of its interface; each call to `heartbeat` is reduced to an increment of a per-task counter. The aggregation of per-task counters divided by the time elapsed from the first heartbeat emission gives the heart rate.

HEART RATE The heart rate of a group is the number of heartbeats issues per time unit by the tasks belonging to the group; we measure the heart rate in heartbeats/s.

Both the performance measurements and the SLO specified by the user make use of the aforementioned performance metric, which is a general throughput metric. Let us recall the *x264* video encoder example; since there is a 1:1 mapping between a frame and a heartbeat, the heartbeat/s performance metric become a frames/s performance metric, which is much more understandable for both applications' developers and users than any other machine-specific performance metric.

The generality of the proposed performance metric does not hurt its usability; in fact, most commercial workloads such as the SPECjbb [44], SPECweb [45], and TPC [46] benchmarks reports their performance in terms of a specific throughput metric that can be easily mapped to the proposed general throughput metric. Some of the scientific and emerging applications belonging to common benchmark suites like the SPEC CPU [47], SPEC OMP [48], Splash-2 [49], and PARSEC [50, 42] may be more difficult to instrument. However, previous work [21, 8, 9] show this is a possible step to accomplish.

2.3.2 *Development*

The performance monitoring infrastructure discussed in this section fulfills and leverages the aforementioned requirements and definitions, respectively. It allows to monitor the heart rate of groups of tasks, where each group usually (but not necessarily) represents an application. Because of this reason, we called the performance monitoring infrastructure the Heart Rate Monitor (HRM). HRM is an active performance monitoring infrastructure[1] featuring a producer/consumer model [26]. HRM achieves ease of use by inheriting a subset of *Application Heartbeats* API. Generality comes from avoiding any machine-specific or operating system-specific functionality; this allows to implement the producer/consumer design and the API by leveraging the POSIX API and nothing more. Once again, the producer/consumer design helps getting high availability; the HRM implementation we proposed [8] places most the consumer logic within the Linux kernel,[2] thus allowing both the kernel- and user-space to access all the performance measurements collected by the performance monitoring infrastructure. Meaningfulness is a gift coming from the use of work-related performance metrics, which are of use for both applications' developers that have the knowledge to correctly instrument

1 Active means the performance monitoring infrastructure requires either manual, semi-automatic, or automatic instrumentation of applications.

2 The very same implementation of HRM that we use on top of the GNU/Linux operating system was successfully ported within the FreeBSD kernel to be used on top of the FreeBSD operating system with minimal effort [12, 9].

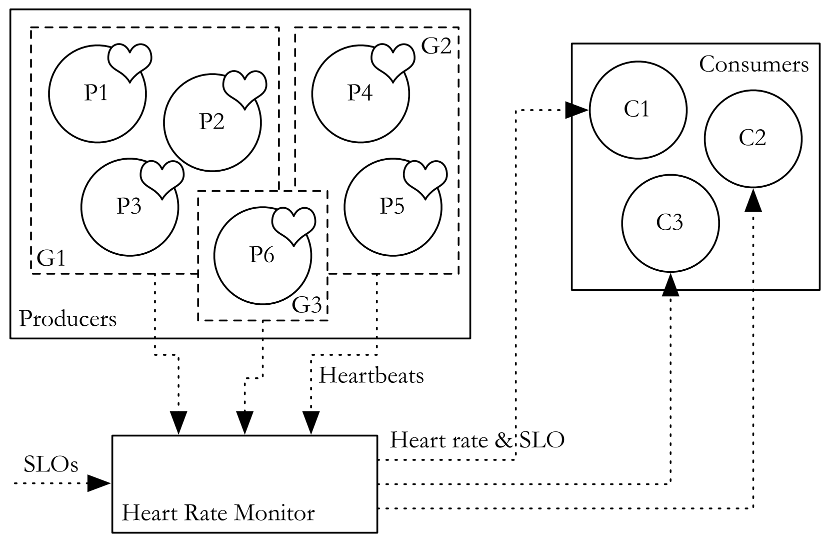


Figure 6.: Black box view of HRM. Instrumented applications represented by groups of tasks issue heartbeats that are collected by the performance monitoring infrastructure, which aggregates them in the form of heart rates. Consumers read the heart rates and user-stated SLOs through the performance monitoring infrastructure.

their software and users that can easily state SLO. The use of work-related performance metrics also grants accuracy for multi-thread applications [33]. Performance comes by moving most of the logic from the applications (*i.e.,* producer side) to the performance monitoring infrastructure (*i.e.,* consumer); this is in net contrast with the *Application Heartbeats* reference implementation. Figure 6 display a rough structure of the ecosystem around HRM.

user-space We implemented the producer/consumer model at the very base of the design of HRM by splitting it across the user- and kernel-space boundary. The user-space side of HRM is implemented by means of a shared object (*i.e.,* a library, *libhrm*) that must be linked against applications (*i.e.,* producers) and to monitors (*i.e.,* consumers) whenever they are implemented in user-space [51, 10]. Table 2 reports a slightly stripped version of the API. A call to `monitor_attach()` allows the current (user-space) task to attach to a group,

Table 2.: HRM API (stripped)

call	description
`monitor_attach()`	attach the current task to a group
`monitor_detach()`	detach the current task from a group
`monitor_set_objective()`	set the SLO
`monitor_get_objective()`	get the SLO
`monitor_get_performance()`	get the performance measurement
`heartbeat()`	signal execution progress

which in the latest version of HRM is identified by a unique string; the task can either be attached as a producer or as a consumer. This function involves many user- to kernel-space boundary crosses to setup data structures, memory mappings, and aggregation timers. Once the virtual memory is shared between the kernel- and user-space a task attached as a producer (*i.e.,* one that is trusted by the user) can set the SLO making it available to both kernel- and user-space monitors by means of a call to `monitor_set_objective()`. Systems' and applications' developers can come to an agreement on how to state SLOs (*e.g.,* minimum throughput [8], minimum/maximum throughout [12, 9, 51, 10], minimum/maximum throughout over a sizeable moving average, etc.) since HRM treats everything as opaque data. A call to `monitor_get_objective()` retrieves the stated SLO in both kernel- and user-space. HRM provides both long- and variable-term performance measurements. The long-term performance measurement aggregates the heartbeats issued across the whole execution of an application. Conversely, variable-term performance measurements aggregate the heartbeats issues across a sizeable time window so as to catch even short-term (*e.g.,* few milliseconds) trends. Tasks attached to groups as producers are those in charge to issue calls to `heartbeat()`; this call issues a heartbeat and signal that the task reached a point of interest in the execution.

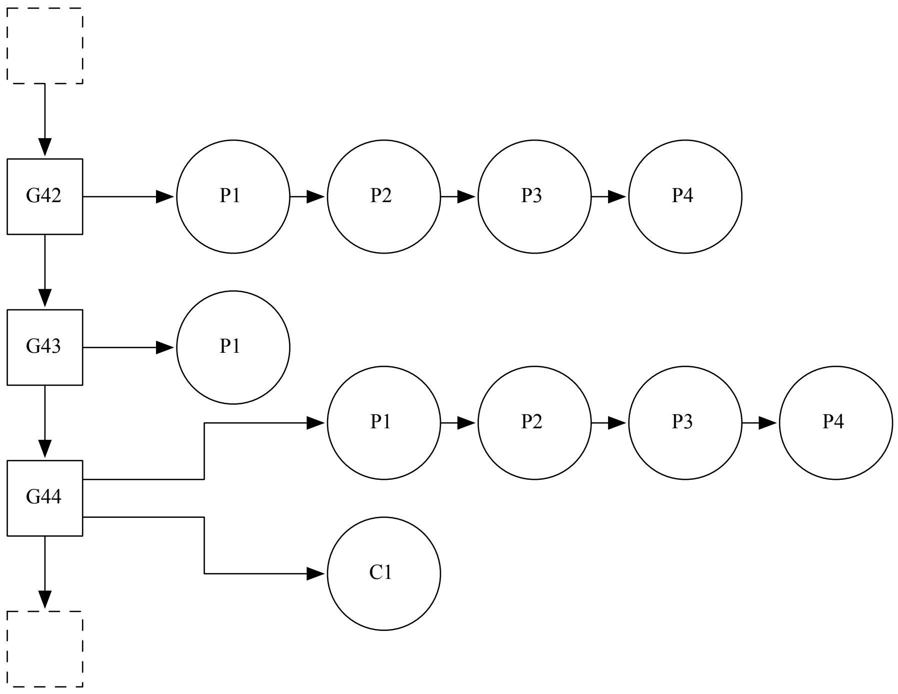


Figure 7.: Buddy-like data structure backing the kernel-space implementation of HRM. Each group (*i.e.,* buddy) contains a set of producer tasks and a set of consumer tasks.

KERNEL-SPACE The kernel-space side of HRM is implemented as an extension of the Linux kernel, tough a similar implementation is available for the FreeBSD kernel, and can be considered the core of the performance monitoring infrastructure. Later on in this chapter we will showcase a complete (*i.e.,* producer/consumer) user-space implementation of HRM; this implementation trades availability for generality like the *Application Heartbeats* reference implementation.

The basic data structure behind proposed implementation of HRM is a buddy-like data structure [52]; each buddy represents a group of tasks. Under the hood, the buddy-like data structure is a linked list of linked lists as shown in Figure 7; in this case, group 42 contains 4 producer tasks of a parallel application, group 43 contains a single producer task, while group 44 contains 4 producer tasks and a single consumer task. The absence of consumers may indicate that the kernel is monitoring the application or that the application apply self-monitoring and self-adaptation [53]. The whole data structure is protected by a global spinlock to guarantee consistency when modifying the buddy-like data structure (*i.e.,* group addition and deletion). Common operations involving a single group does not need to acquire the global spinlock; instead, they need to acquire the read/write spinlock (rwlock) of the sub linked list of interest (*i.e.,* either producers or consumers). The rwlock helps scalability since the number of read and write operations is unbalanced in favor to the former; thus, concurrent read can proceed in parallel while write serialize the execution. Each group features a set of memory pages to store the per-task counters, the history of the aggregations of per-task counters to compute variable-term performance measurements, and the SLO specification. The number of per-task counters and the depth of the history, which constrains the computable variable-term performance measurements, are compile-time parameters. These memory pages are shared across the kernel- and user-space boundary and possibly across many address spaces when an application employs Process-Level Parallelism (PLP).

As reported previously in this section, *libhrm* provides both producers and user-space consumers with the ability of attaching and detaching (among the others) from groups of tasks. The access from user- to kernel-space leverages the process information pseudo file system (procfs).[3] `write(2)` operations on `/proc/$PID/task/$TID/hrm_[producer|consumer]_group` attaches (detaches) the task identified by `$TID` from a certain group while `mmap(2)/munmap(2)` operations on `/proc/$PID/task/$TID/hrm_[producer|consumer]_counters`, `/proc/$PID/task/$TID/hrm_[producer|consumer]_history`, and `/proc/$PID/task/$TID/hrm_[producer|consumer]_objective` share (unshare) memory pages across the user- and kernel-space boundary.

The kernel-space side of HRM also export some of the API calls available inside the user-space side: *libhrm*. These calls are necessary to support the implemen-

3 The extension of the FreeBSD kernel makes use of virtual devices to achieve the same functionality.

Table 3.: HRM API availability

call	producer	consumer	kernel	user
`monitor_attach()`	✓	✓		✓
`monitor_detach()`	✓	✓		✓
`monitor_set_objective()`	✓			✓
`monitor_get_objective()`	✓	✓	✓	✓
`monitor_get_performance()`	✓	✓	✓	✓
`heartbeat()`	✓			✓

tation of kernel-space consumers, one of the distinctive characteristics of HRM with respect to the *Application Heartbeats* reference implementation. Table 3 sums up which API calls are or are not accessible to producers/consumers and the kernel- and user-space.

2.3.3 *Cache Craftiness for Performance*

An objective for a performance monitoring infrastructure is to affect as little as possible the performance of the applications it is monitoring. The *Application Heartbeats* reference implementation is not capable of doing so as a consequence of serialization it performs for issuing heartbeats and compute the performance measurements and the system call it requires to keep track of the elapsed time.

The design of HRM allows the performance monitoring infrastructure to keep its overhead as low as possible, thus achieving one of its main objectives. The implementation of HRM employs per-task counters to issue heartbeats, thus avoiding serialization on the critical paths of applications; in addition, each per-task counter is aligned to the cache line size so as to evade problems due to false sharing. Later in this section we will show how much false sharing could impact on the raw performance HRM to demonstrate how helpful are this kind of cache craftiness with multicore processors.

Table 4.: HRM per-task counters memory page organization

offset	content	description
`0x000`		miscellaneous fields for housekeeping (*e.g.,* `$TID`, use & active flags, etc.)
`0x010`	`uint64_t counter`	heartbeats counter
`0x018`		padding for cache line alignment
`0x040`		miscellaneous fields
`0x050`	`uint64_t counter`	heartbeats counter
`0x058`		padding
⋮	⋮	⋮
`0xFC0`		miscellaneous fields
`0xFD0`	`uint64_t counter`	heartbeats counter
`0xFD8`		padding

Table 4 reports per-task counters memory page organization. The table uses the x86-64 architecture as an example since the size of the memory page is 4 kB[4]. and the size of each cache line (separated by horizontal lines in the table) is 64 B.

2.3.4 *Computation of the Performance Measurements*

The access to both history of the aggregations of the per-task counters and to the SLO are certainly much rarer than those performed to the per-task counters (*i.e.,* at the hundreds of milliseconds granularity). Because of this, there is no need for extra care and cache craftiness when laying out the data structure inside the memory page.

4 We did not explore the use of huge memory pages of either 2 MB or 1 GB since we never exceeded 6-thread per application in our experiments. In addition, the shift from standard to huge memory pages would be required only in presence of increased Translation Lookaside Buffer (TLB) pressure.

Each group is provided with a high-resolution timer that is in charge of performing the aggregation (*i.e.,* sum) of the per-task counters at a constant pace while holding the proper rwlock. When it finishes the aggregation process, the high resolution timer stores the result within a circular buffer maintained inside the history memory page.

The use of a high resolution timer instead of a system call as it is in the *Application Heartbeats* reference implementation improves performance since it avoids plenty of the user- to kernel-space traps.

Consumer tasks get performance measurements both in kernel- and user-space by issuing a call to `hrm_get_performance(int depth)`. Whenever the depth parameter is 0, the API call returns the long-term performance measurement whose value is computed according to Equation (1). If the depth parameter holds a value greater than 1 and less than the maximum depth previously mentioned, the API call returns a variable-term performance measurement over the last `depth` aggregations of the per-task counters. Equation (2) reports the mathematical formulation of the measurement.

$$r(k) = \frac{\sum_i counters_i(k)}{k} \tag{1}$$

$$r(k) = \frac{\sum_i counters_i(k) - counters_i(k-d)}{k-d} \tag{2}$$

Within both the equations, r is the performance measurement, $counter_i$ represent the per-task counter of the i-th task, k is the sampling interval, and d is the value stored inside the `depth` parameter.

With the implementation proposed HRM offloads the computation of performance measurements to consumer tasks while the same job is accomplished by the application itself with the *Application Heartbeats* reference implementation.

2.4 evaluation

This section evaluates HRM and, in particular, it is focused on answering the following questions:

1. is HRM a high performance solution with respect the state of the art, the *Application Heartbeats* reference implementation?
2. is HRM easy to use for instrumenting emerging real-world applications employing various parallel model (*e.g.,* fork & join, worker pools, pipeline, spawn & kill, etc.)?
3. is HRM a low overhead on emerging real-world applications?

2.4.1 *Evaluation Platforms and Configurations*

We evaluated HRM on three different evaluation platforms. The first is a workstation with an Intel Core i7-870 Processor and 4 GB of DDR3-1066. The processor features 4 cores operating at 2.97 GHz and sharing 8 MB of Last-Level Cache (LLC). We disabled the Enhanced Intel SpeedStep Technology and the Intel Turbo Boost Technology to prevent the processor entering P-states with clock frequency lower or higher than the nominal when a subset of the cores is executing. We disabled the Intel Hyper-Threading Technology (HTT) to simplify our analysis. This evaluation platform is representative of a commodity CMP.

The second evaluation platform is a workstation with an Intel Pentium D Processor 820 and 2 GB of DDR2-800. The processor features 2 cores operating at 2.80 GHz, each one with 1 MB of LLC. We believe this evaluation platform is representative of a commodity multi-socket system since the 2 cores live on separate dies communicating through the system bus even through they are on the same socket.

The third is a Dell T3500 workstation with an Intel Xeon W3670 Processor and 12 GB of DDR3-1066. The processor features 6 cores operating at 3.20 GHz and sharing 12 MB of LLC. We disabled the Enhanced Intel SpeedStep Technology and the Intel Turbo Boost Technology to prevent the processor entering P-states with clock frequency lower or higher than the nominal when a subset of the cores is executing. We disabled the Intel Hyper-Threading Technology (HTT) to simplify our analysis. This evaluation platform is representative of a commodity CMP.

We configured Debian 6.0 to boot with the Linux kernel 2.6.35 enhanced with HRM. HRM was configured to support at most 64 tasks per group (*i.e.,* one memory page) and a depth of the variable-term performance measurement of up to 1024. High resolution timers were scheduled to aggregate the per-task counters every 100 ms.

We answered the questions reported above through a hand-crafted microbenchmark and the PARSEC 2.1 benchmark suite [50, 42]. The microbenchmark is useful to collect raw performance numbers for both HRM and the *Application Heartbeats* reference implementation while the complete benchmark suite provides a set of emerging real-world applications with different parallel models to asses the ease of use of HRM and its low overhead.

2.4.2 *High Performance*

We evaluated the raw performance of the two performance monitoring infrastructure (*i.e.,* HRM and the *Application Heartbeats* reference implementation through microbenchmarking. The microbenchmark allows specifying the level of parallelism (*i.e.,* the number of tasks to fork) and the number of heartbeats to issue. The hotspot of the microbenchmark is a tight loop issuing heartbeats as simple as the one shown in Figure 8 where `ntasks` is the number of tasks forked by the microbenchmark and `nheartbeats` is the number of heartbeats each task must issue. Because of this setting, the performance measurement

```
nheartbeats = 1000000 / ntasks;
for (int i = 0; i < nheartbeats; ++i)
        heartbeat();
```

Figure 8.: Tight loop issuing heartbeats within the microbenchmark.

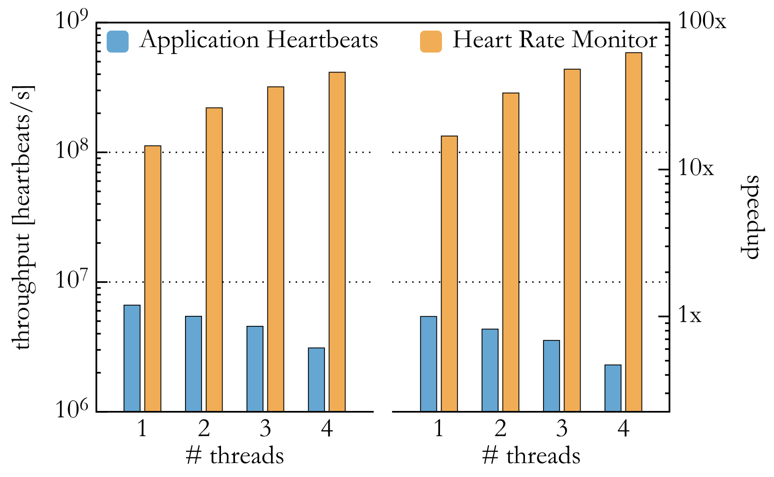


Figure 9.: Throughput (higher is better) and speedup (higher is better) of the microbenchmark—1 to 4 threads—with both the *Application Heartbeats* reference implementation and HRM. The speedup baseline is the microbenchmark—1 thread—result with the *Application Heartbeats* reference implementation. The plot reports arithmetic averages over a thousand executions with the 95 % confidence interval less than 1 % of the measurement.

(*i.e.*, the heart rate) of this microbenchmark quantifies the overhead; the higher the performance measurement, the lower the overhead.

Figure 9 shows the throughput and the speedup of the microbenchmark with both the *Application Heartbeats* reference implementation and HRM executing on the first evaluation platform.

HRM is at least an order of magnitude faster than the *Application Heartbeats* reference implementation and we did not observe any difference in terms of raw performance between forking threads (results shown in the plot) or processes. The speedup shown on the right side of the figure clearly shows that

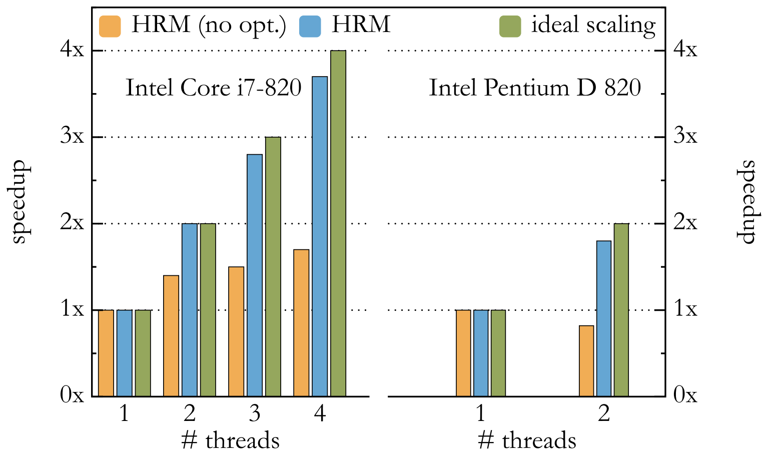


Figure 10.: Speedup (higher is better) of the microbenchmark—1 to 4 threads and 1 to 2 threads—with HRM without and with cache craftiness. The speedup baseline is the microbenchmark—1 thread—result with the corresponding configuration. The ideal speedup is the number of threads (*i.e.*, cores). The plot reports arithmetic averages over a thousand executions with the 95 % confidence interval less than 1 % of the measurement.

the raw performance with the *Application Heartbeats* reference implementation decreases while the number of tasks increases, which is counterintuitive but common for certain applications executing on multicore processors [4]. Conversely, the raw performance with HRM increases with the number of tasks, which is the behavior most people would expect from an application.

Figure 10 shows in greater details the scalability of HRM without and with cache craftiness executing on both evaluation platforms.

HRM without cache craftiness (see yellow bars) displays a better behavior than the *Application Heartbeats* reference implementation, which was not scalable at all. However, this version of HRM is still far from the ideal scalability (see green bars) on the first evaluation platform, which makes use of the shared LLC to exchange cache lines among cores, and does not even scale on the second evaluation platform, which employs the system bus to exchange cache lines among cores. Conversely, HRM with cache craftiness approaches the ideal scalability on both the evaluation platforms (see blue bars), thus showing the relevance of a multicore processors-aware design and implementation.

2.4.3 *Ease of Use*

We analyze the instrumentation of 11 out of 13 multi-thread applications from the PARSEC 2.1 benchmark suite [50, 42] employing diverse parallel models, which allows to show the ease of use of *libhrm*.

We focus on the following applications: *blackscholes*, *bodytrack*, *canneal*, *dedup*, *facesim*, *ferret*, *fluidanimate*, *raytrace*, *streamcluster*, *swaptions*, and *x264*. These 11 multi-thread applications can be grouped in four categories according to their parallel model:

- *category 1—blackscholes*, *canneal*, *fluidanimate*, *streamcluster*, and *swaptions* use "fork & join" of workers;
- *category 2—bodytrack*, *facesim*, and *raytrace* leverage pools of workers running different jobs in parallel;
- *category 3—dedup* and *ferret* use a pipeline with a pool of workers per stage;
- *category 4—x264* employs "spawn & kill" of workers to realize a virtual pipeline.

For each category, we give additional details regarding the structure and instrumentation of one application.

Applications in *category 1* are straightforward to instrument, as they use a simple parallel model. The sequence diagram in Figure 11 shows the structure and instrumentation of these applications. The main thread of the applications is responsible for forking (*i.e.,* `pthread_create(3)`) the worker threads and joining them (*i.e.,* `pthread_join(3)`) when they exit (*i.e.,* `pthread_exit(3)`). The first worker thread attaches to (and implicitly creates) the group, and it

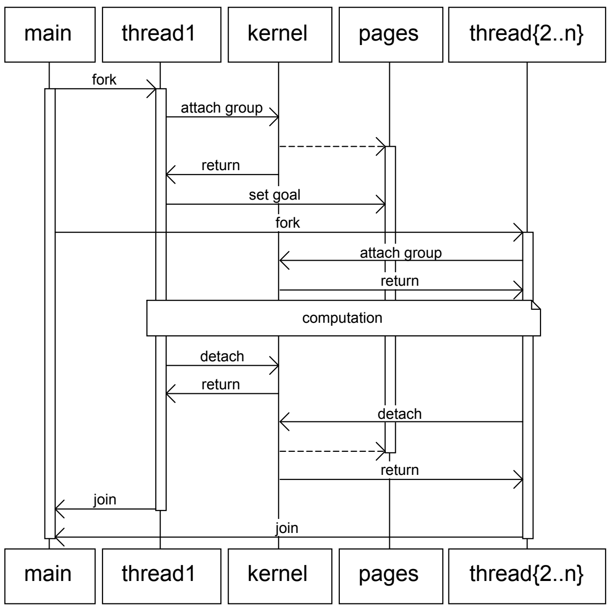


Figure 11.: Structure of the applications in *category 1*.

sets the SLO; the other worker threads attach to the group and, just like the first worker thread, start their computation. Figure 12 visualizes the typical structure of computation of a worker thread, which runs in a loop terminating a unit of work, issuing a heartbeats. When the application is terminating, before re-joining, the worker threads detach from the group.

Applications in *category 2* use a pool of worker threads running parallel kernels; therefore, none of the threads completes a unit of work alone. Figure 13 represents the common structure of these applications. The main thread, which acts as a dispatcher, is the first to attach to the group. Due to the structure of these applications, which is represented in Figure 14, the main thread issues heartbeats. However, we still attach all the worker threads to the group to make resource allocation policies and mechanisms aware that they are relevant for the execution.

Applications of *category 3* employ many pools of worker threads organized in a pipeline. In these applications, the main thread is responsible for forking the pools of worker threads and waiting for them to join upon application completion. All the forked threads attach to the group (the first thread automatically creates the group and sets the SLO); Figure 15 shows the general structure of these applications. The worker threads contained in the n-th pool (*i.e.,* the last stage of the pipeline) are the ones committing each unit of work and are responsible for issuing heartbeats, as Figure 16 illustrates.

The last category (*i.e., category 4*) contains only one application, namely *x264*. *x264* creates a virtual pipeline based on a "spawn & kill" parallel model, which makes the instrumentation straightforward. Figure 17 illustrates the structure of the instrumentation of *x264*. The main thread is responsible for creating and attaching to the group. Figure 18 focuses on the computation phase: the main thread spawns many different worker threads that re-join when their computation ends. The main thread maintains the notion of advancement (*i.e.,* encoding of frames in *x264*); thus, it is responsible for issuing heartbeats. Just as for applications in *category 2*, we still attach all the worker threads to

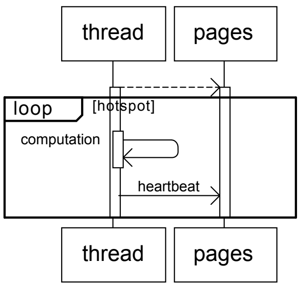


Figure 12.: Computation the applications in *category 1*.

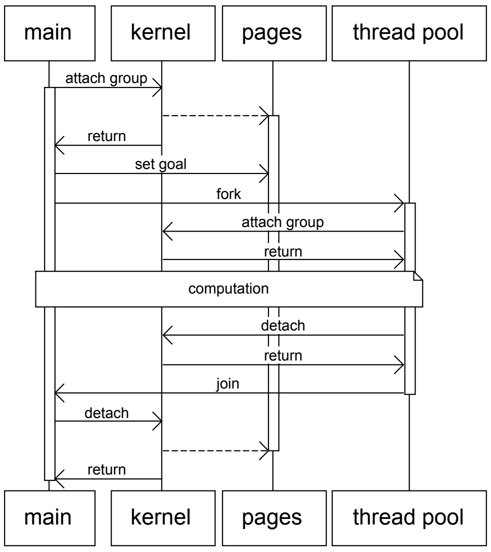


Figure 13.: Structure of the applications in *category 2*.

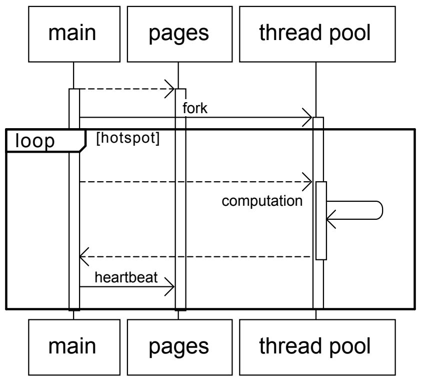


Figure 14.: Computation of the applications in *category 2*.

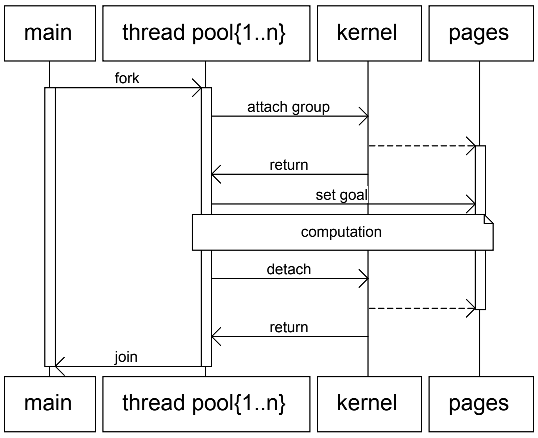


Figure 15.: Structure of the applications in *category 3*.

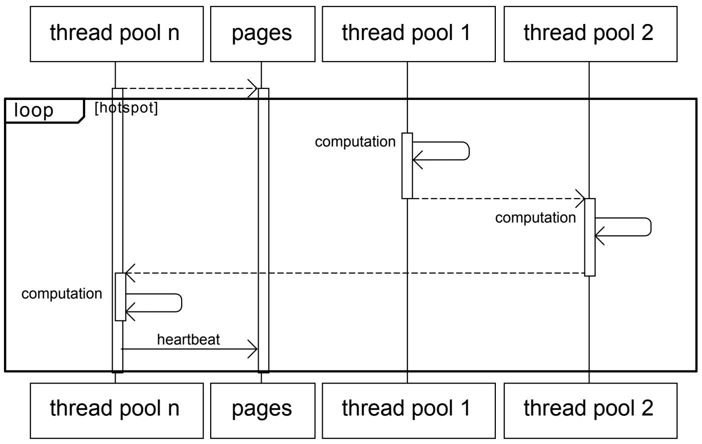


Figure 16.: Computation of the applications in category 3.

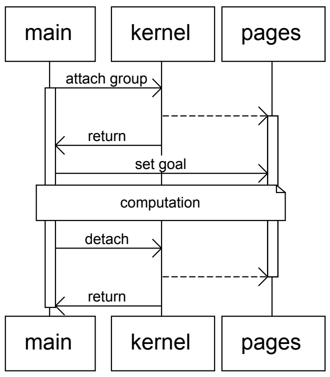


Figure 17.: Structure of the applications in *category 4*.

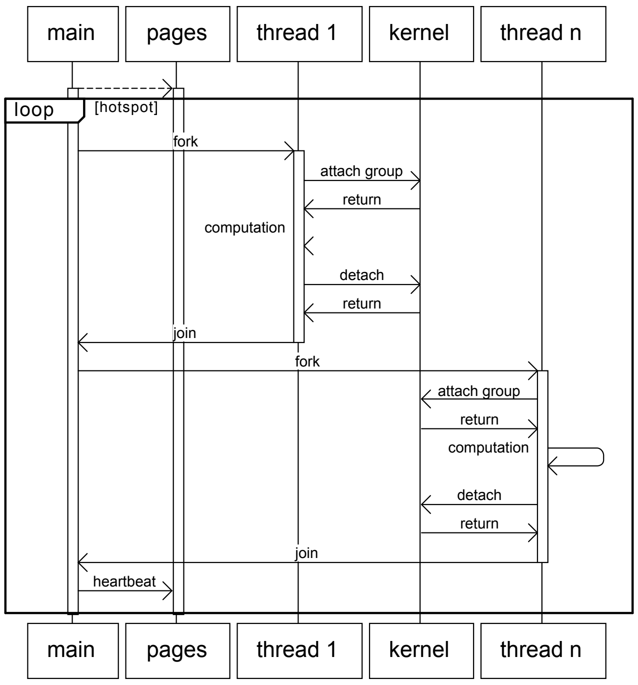


Figure 18.: Computation of the applications in category 4.

Table 5.: Comparison between *vanilla* and instrumented applications from the PARSEC 2.1 benchmark suite

category	application	*vanilla* avg. [ms]	*vanilla* std. [ms]	instrumented avg. [ms]	instrumented std. [ms]	overhead
1	*blackscholes*	68731.67	1998.33	68902.53	221.21	0.25 %
	canneal	96405.94	1846.36	96913.76	488.74	0.53 %
	fluidanimate	95785.44	627.38	96077.83	103.19	0.31 %
	streamcluster	147536.15	2393.04	147460.57	333.19	-0.05 %
	swaptions	75308.29	308.39	75508.16	249.35	0.27 %
2	*bodytrack*	52849.39	412.03	53732.33	878.61	1.67 %
	facesim	145175.15	2256.19	145408.80	787.26	0.16 %
	raytrace	124036.75	901.47	124441.34	750.43	0.33 %
3	*dedup*	33509.96	955.21	34448.43	1187.31	2.80 %
	ferret	113626.21	527.36	114106.41	218.52	0.42 %
4	*x264*	32657.13	252.06	32713.23	255.53	0.17 %

the group to inform resource allocation policies and mechanisms about their relevance.

2.4.4 *Low Overhead*

We evaluate the overhead of HRM on all the multi-thread applications we instrumented from the PARSEC 2.1 benchmark suite [50, 42].

Table 5 reports average execution times and their standard deviations over 100 consecutive execution of unmodified (*i.e., vanilla*) and instrumented applications with the native inputs. The table also reports the overall runtime impact (*i.e.,* overhead). Experimental results where collected on the third evaluation platform.

The highest runtime impact we measured is 2.80 % for *dedup*; with the exception of *x264*, higher runtime impacts (*e.g., bodytrack* and *dedup*) coincide with short execution times and we argue this is due to “non-amortized” costs of creating the group and attaching worker threads, which are the most expensive operations with HRM. According to experimental results we can state that HRM is efficient and imposes negligible runtime impact; in fact, in some of the applications (*e.g., streamcluster*) the runtime impact becomes "negative"—though remaining within the confidence of the measurement.

2.5 SUMMARY

The design and implementation of HRM addresses all the requirements we pointed our in Section 2.2. The evaluation reported in Section 2.4 demonstrates our claims regarding the performance monitoring infrastructure: high performance, ease of use, and low overhead.

Despite being born as a performance monitoring infrastructure, HRM is a general-purpose solution to acquire throughput measurement and we believe it is suitable in a wide variety of contexts. For instance, HRM was used to measure contention over spinlocks by instrumenting a synchronization library to issue a heartbeat each time a task (*i.e.,* producer) spins over an already acquired spinlock. Each task planning on acquiring and releasing the spinlock can attach a the spinlock group. In this scenario, the higher the heart rate, the higher the contention, and the objective of a resource allocation policy could be minimizing the heart rate by driving a resource allocation mechanism to move tasks contending the same spinlocks on the same socket to leverage share LLC cache line exchanges [43].

This chapter presented various contributions from published work [8, 12, 9] and touched contributions from unpublished work [43].

3 ADAPTIVE PERFORMANCE MANAGEMENT

In the last decade computer architecture crossed a critical junction and so did the whole computing industry shifting from single to multicore processors. The introduction of Chip-Multiprocessors (CMPs) requires better support from system software: hypervisors, operating systems, and runtimes. System software must be able to manage emerging computer architectures achieving the same predictable performance single-thread applications used to have on singlecore processors. In addition, the growing importance of non-functional requirements (*e.g.,* performance, energy consumption, etc.) imposes system software to support the specification of Service-Level Objectives (SLOs) and the resource allocation mechanisms to meet them. Finding the proper resource allocations to yield the specified SLOs for multi-program workloads can be cumbersome; because of this, automatic resource allocation policies should close the loop by driving the available mechanisms. System software, operating systems in this specific case, are the perfect test bed for experimenting with various resource allocation policies and mechanisms as they find themselves in a favorable position in the hardware/software execution stack.

In this chapter we present various solutions to manage performance following the design paradigm dictated by the autonomic control loop: Observe-Decide-Act (ODA). First, we present Metronome: a Central Processing Unit (CPU) bandwidth allocation policy and mechanism implemented inside the Completely Fair Scheduler (CFS) (*i.e.,* the general scheduling class of the Linux kernel hierarchical scheduler). Second, we present Metronome++: a CPU core allocation policy and mechanism implemented implemented on top of Linux kernel.

3.1 introduction

The failure [54] of Dennard's scaling law [1] led chip manufacturers to slowly abandon the superscalar microarchitectures that characterized singlecore processors till the beginning of the last decade. Even though the Moore's law [3] is still supporting the exponential increase of transistors' count, the growth of clock frequency lost its pace just like the decrease of the supply voltage did recently. This brought to the lack of success of highly superscalar microarchitectures like the Intel NetBurst powering the Intel Pentium 4 Processors. As a consequence, single-thread performance is not growing at historical rate anymore and we are even assisting to the diffusion of simpler microarchitectures supporting in-order execution instead of out-of-order execution like with the Intel Atom Processors or the International Business Machines (IBM) POWER7 Processors.

Multicore processors gradually replaced singlecore processors and the industry is already shifting towards manycore processors. Multicore and manycore processors are an ensemble of cores, possibly characterized by a simpler microarchitecture, living on a single piece of silicon; because of this, they are also known as CMPs. The existence of a number of active applications in computing systems and the development of multi-thread flavors of these applications—possibly massively parallel—benefit from the growing cores' count broadening the spectrum of interesting execution scenarios. The focus of research moved from executing a single-thread application on a singlecore processor to:

1. execute a multi-thread application on a multicore processor;

2. execute multiple single-thread applications on a multicore processor;

3. execute multiple multi-thread applications on a multicore processor;

in every research community, from computer architecture to compilers and systems (*i.e.,* operating systems, distributed systems, etc.). Whenever we found ourselves analyzing the second and third execution scenarios we talk about multi-program workloads execution.

To allow the reader to better understand the relevance of this execution scenarios, we should notice that the advent of cloud computing encourages the both the second and third one. Cloud computing folks have a specific term other than multi-program workloads execution that is: consolidation. Consolidation means concurrently executing multiple applications wrapped inside Virtual Machines (VMs) or other kinds of resource containers [55] on the same physical machines, thus possibly sharing resources such as CMPs.

From a computer architect standpoint, the paradigm shift from single-thread execution exploiting Instruction-Level Parallelism (ILP) to multi-thread and multi-program execution exploiting Thread-Level Parallelism (TLP) meant higher efficiency and utilization: a win-to-win game. From a developer perspective, a lot of complexity rose, which now burdens both systems' and applications' developers. Where once applications had no operating points, they can now execute with 1 to n threads whose placement on a CMP and especially across CMPs can greatly affect their efficiency [56]. In addition, singlecore processors allowed changing their operating points, for example through duty cycle modulation or Dynamic Voltage and Frequency Scaling (DVFS), affecting the execution of a single application. With CMPs, cores living on the same piece of silicon may or may not share operating points [57, 11] and different applications may require different configuration [58, 6] to attain, for example, the highest overall performance/W, making these kind of optimization cumbersome.

Traditionally, system software (*i.e.*, hypervisors, operating systems, and runtimes) is in charge of providing a Hardware Abstraction Layer (HAL) over the bare-metal. We believe system software must retain its privileged role, thus systems' developers should bear most of the burden they share with applications' developers since they have the complete picture of the hardware/software execution stack. System software must be able to deliver the same predictable performance single-thread applications used to experience on singlecore processors and, at the same time, must take into account the emerging non-functional requirements such as performance, energy and power consumption, temperature, reliability, availability, etc. In the remainder of this chapter we will go through different subsystems we design and implemented to extend commodity operating systems, the system software of choice, so as to improve the support for multi-program workloads execution on CMPs under SLOs.

This chapter reports the following contributions:

- we describe Metronome, a CPU bandwidth allocation policy and mechanism implemented inside CFS to time-share CMPs' resources among the applications of a multi-program workload accounting for their SLOs;

- we describe Metronome++, a CPU core allocation policy and mechanism implemented on top of the Linux kernel to space-share CMPs' resources among the applications of a multi-program workload accounting for their SLOs and characteristics (*i.e.*, scalability);

Both Metronome and Metronome++ coupled with Heart Rate Monitor (HRM) are two alternative adaptive performance management solutions and two flavors of the Autonomic Operating System (AcOS) [9], the macro project this dissertation fits into.

The remainder of this chapter is organized as follows. Section 3.2 and Section 3.4 describe the design and implementation of Metronome and Metronome++, respectively. Section 3.3 and Section 3.5 do the same for their evaluations. Section 3.6 discusses a set of common related work while Section 3.7 summarizes the chapter.

3.2 DESIGN AND IMPLEMENTATION OF METRONOME

In the context of system software (*i.e.,* hypervisors, operating systems, and runtimes) the task scheduler has always been the component in charge of determining the allotment of CPU bandwidth to the runnable tasks.

The choice of the policy or policies ruling the scheduling infrastructure can highly impact the behavior of the computing system, thus different policies applies in different scenarios. A possible classification follows:

- batch scenarios are characterized by a huge amount of jobs to be completed sequentially, without user impatiently waiting for interacting with a specific task. This scenarios are typical of some servers, datacenters, or supercomputers;
- interactive scenarios are characterized by one or more users who want to interact with some task. This scenarios are typical of desktops and mobiles.
- real-time scenarios in which computing systems run time-constrained tasks. We go one level down in the classification by identifying:
  - hard real-time scenarios where deadlines are strict and can never be missed by any of the runnable task; these scenarios are the norm in avionics and similar environments;
  - soft real-time scenarios where runnable tasks are allowed to miss a few deadlines even though they tend to guarantee a certain Quality of Service (QoS), expresses as the maximum percentage of deadlines a runnable task is allow to miss. A notable example of a soft real-time task is video decoding, which should keep up with a certain rate to guarantee a flawless video playing experience.

We usually refer to those scheduling infrastructures that do not provide any kind of real-time guarantee as best effort solutions.

Usually, each of the scenarios reported above has its own policy. Diverse policies usually share some common objectives and then specialize to reach some specific objectives that depends on the scenario they address. The set of objectives that are common among the different policies is:

- fairness, which means that tasks with the same priority should receive the same amount of CPU bandwidth;
- balance, which denotes the ability of spreading jobs equally among the available CPU cores.

The set of objectives that are specific to the different scenarios are:

- for batch scenarios:
  - turnaround time minimization, by making the execution time of each job as short as possible, thus maximizing the overall throughput;
  - CPU use maximization, which means keeping the CPU as busy as possible.
- for interactive scenarios:
  - response time minimization, by reducing the perceived latency;
  - proportionality, by giving all users the expected share of CPU bandwidth for their tasks;
- for real-time scenarios:
  - missed deadline minimization, by respecting QoS within soft real-time scenarios and by reducing the number to 0 within hard real-time scenarios;
  - predictability, being able to deterministically say in advance whether a deadline can be enforced or not, thus achieving the same execution from time to time.

The above classification [59] is very neat in assigning specific objective to each scenario. However, we believe some computing systems present slightly blurred scenarios where mixes of the aforementioned objectives are much appreciated. In fact, the trade-off among the specific objectives can usually be statically tuned in commodity operating systems. For example, the Linux kernel comes with a hierarchical scheduling infrastructure where scheduling classes can be plugged-in and out at compile time to adhere to the real-world scenario [37]. The default configuration features a simple scheduling class whose aim is to maximize the overall throughput[1] and a fairly advanced scheduling class whose main objectives are response time and fairness.[2] A scheduling class implementing the Earliest Deadline First (EDF) scheduling policy to handle both hard and soft real-time tasks is also available: `SCHED_DEADLINE` [60]. The Linux kernel allows changing the amount of CPU bandwidth each scheduling class receives in order to better handle the trade-off according to the real-world scenario. Alternative solutions employ two-level scheduling infrastructure to achieve similar goals [27, 29, 30, 28].

Because of its high impact on the behavior of computing systems, the scheduling infrastructure represents a suitable component in which adaptive capabilities, and hence adaptive performance management, can be embedded. Metronome (formerly known as Performance-Aware Fair Scheduler (PAFS) [8]) couples a resource allocation policy and a mechanism that extend the general-purpose scheduling class within the hierarchical scheduling infrastructure of recent Linux kernels: the Completely Fair Scheduler (CFS). The idea behind Metronome is to re-define the concept of fairness to support applications (*i.e.*, groups of tasks, see Chapter 2) which are bound to user-stated SLOs. This change to the CFS scheduling class as the side effect of introducing what we believe is a desirable property for every scheduling infrastructure: predictability, which unfortunately is common only in real-time scheduling infrastructures.

1 This class handles both First In First Out (FIFO) and Round-Robin (RR) tasks; the former run to completion as soon as they get the CPU while the latter can be preempted by other user-space tasks.
2 This scheduling class handles most of the runnable tasks in the computing system and is called the Completely Fair Scheduler (CFS).

3.2.1 *Definitions*

The current scheduling infrastructure supporting a hierarchy of scheduling classes that powers the Linux kernel was introduced by Ingo Molnár back in 2007 with the release of version 2.6.23.[3] The design of CFS, which is the general-purpose scheduling class within the Linux kernel was influenced by the Rotating Staircase Deadline (RSDL) scheduling infrastructure, which was a proposal of Con Kolivas that never made into the mainline of the Linux kernel. The main idea behind these scheduling policies is achieving fairness without the need to characterize whether a task is behaving either as a batch one or as an interactive one. The classic concepts of epoch and timeslice were discarded in favor of a simpler but more effective one: the virtual runtime.

VIRTUAL RUNTIME The virtual runtime (*i.e.,* vruntime) of a task is the actual actual runtime (the amount of time spent running) of the task normalized (*i.e.,* weighted) by the task priority and the number of runnable tasks considering their priority. It is measured in nanoseconds.

The vruntime of the running tasks gets updated either at each scheduler tick, whose frequency is a compile time configuration, or when a task yields the CPU core as a consequence of re-schedule request. The `nice(1)` value of the running task is taken into account during the vruntime update operation and it is used to weigh the update value: tasks with a higher priority (*i.e.,* lower nice value) will get a smaller than actual update value while the opposite will happen for tasks with a lower priority. Within CFS tasks are organized in per-CPU core runqueues; each runqueue is implemented through a red-black

3 The previous scheduling infrastructure made use of a more classic multi-queue data structure and a set of heuristics to determine if a task was either behaving as a batch one or as an interactive one. This set of heuristics was the weak spot of the O(1) scheduling infrastructure.

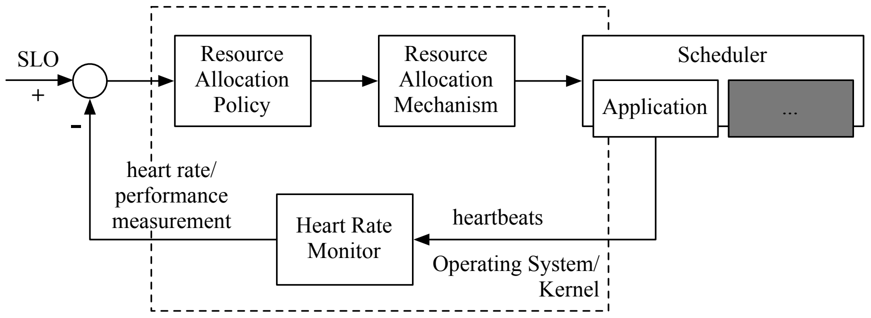

SLO
+
-
Resource Allocation Policy
Resource Allocation Mechanism
Scheduler
Application
...
heart rate/ performance measurement
Heart Rate Monitor
heartbeats
Operating System/ Kernel

Figure 19.: Exemplified system architecture of Metronome. Each application (*i.e.,* ensemble of tasks) attaches to HRM; the resource allocation policy decides whether the "priority" of the application is enough, eventually instructing the resource allocation mechanism to give the application more or less CPU bandwidth.

tree that define a time line of tasks. The ordering key of the time line is the value k reported in Equation (3).

$$k = \text{vruntime}_i - \text{mvruntime} \tag{3}$$

$$\forall j \quad | \quad j \in Q(i) \quad \text{vruntime}_j \geqslant \text{mvruntime} \tag{4}$$

Where i and j are the i-th and j-th tasks in the time line, respectively, and $Q(i)$ is the set of tasks living in the same runqueue as the i-th task. At this point, the scheduling policy becomes dead simple: run the tasks with the lower value of k that is the left-most in the time line.[4]

3.2.2 *Development*

As mentioned earlier in this section, Metronome couples a resource allocation policy and a mechanisms that, together with HRM (see Chapter 2), realize an adaptive performance management extension of a scheduling class of the Linux kernel scheduling infrastructure (see Figure 19). Metronome assigns the CPU

4 We thanks Jones [61] for the material he posted on the design and implementation of CFS, which was very useful to understand the behavior of the scheduling class.

bandwidth to both instrumented applications (*i.e.,* applications instrumented with *libhrm*) and legacy applications boosting the possibility of the former to meet their SLOs while keeping the classic definition of fairness for the latter. At a first glance Metronome could resemble soft real-time scheduling infrastructure; however, we classify Metronome as a best effort solution since it cannot give guarantee on the achievement of SLOs. Hence, Metronome explores the possibility of binding a best effort solution with the ability of driving performance measurements collected through HRM towards SLOs.

The main idea underlying Metronome is to alter the definition of fairness enforce by CFS for those applications instrumented with *libhrm* that provides SLOs; Metronome does so by changing the vruntime update operation. Within CFS the vruntime update operation is carried on according to Equation (5).

$$\text{vruntime}_i(t) = \text{vruntime}_i(t-1) + \Delta_i(t) \quad \forall i, t \quad \Delta_i(t) > 0 \tag{5}$$

Where i is the i-th task, vruntime$_i(t)$ is the vruntime of the task after the update operation while vruntime$_i(t-1)$ is the vruntime of the before the update operation, and $\Delta_i(t)$ is the update value, which must be greater than 0 to avoid starvation. Within Metronome the vruntime update operation is carried on in the same way; however, we substitute $\Delta_i(t)$ with a new update value: $\Pi_i(t)$. Obviously, the relationship between $\Pi_i(t)$ and $\Delta_i(t)$ is as follows:

- if $\Pi_i(t) < \Delta_i(t)$, the i-th task will be advantaged since its vruntime will grow at a slower with Metronome than in would with CFS; thus, it will receive a larger share of the CPU bandwidth;
- if $\Pi_i(t) > \Delta_i(t)$, the i-th task will be penalized since its vruntime will grow at a faster with Metronome than it would with CFS; thus, it will receive a smaller share of the CPU bandwidth;
- if $\Pi_t(t) = \Delta_i(t)$, which is unlikely, Metronome and CFS behave in the exact same way.

The resource allocation policy is a heuristic in which $\Pi_i(t)$ is a function of: the update value $\Delta_i(t)$, the performance measurement and the SLO of the of the group (see Section 2.3.1) the i-th task belongs to. Within Metronome the SLO is a two-level solution with a minimum and a maximum value. $\Pi_i(t)$ is computed according to the piecewise-defined function reported in Equation (6).

$$\Pi_i(t) = \begin{cases} p_1 \cdot \frac{r_{g(i)}(t)}{\bar{r}_{g(i)}} \cdot \Delta_i(t) & r_{g(i)}(t) < \overline{r_m}_{g(i)} \\ \Delta_i(t) & \overline{r_m}_{g(i)} \leqslant r_{g(i)}(t) \leqslant \overline{r_M}_{g(i)} \\ p_2 \cdot \frac{r_{g(i)}(t)}{\bar{r}_{g(i)}} \cdot \Delta_i(t) & r_{g(i)}(t) > \overline{r_M}_{g(i)} \end{cases} \tag{6}$$

Where p_1 and p_2 are parameters to modulate the strength of the corrective action, $g(i)$ is the group at which the i-th task belongs to, $r_{g(i)}(t)$ is the performance measurement, $\overline{r_m}_{g(i)}$ and $\overline{r_M}_{g(i)}$ are the minimum and maximum values as they are defined in the SLO of the group, and $\bar{r}_{g(i)}$ is the average between the two (we expect the interval not to be wide to avoid excessive oscillations within the SLO).[5]

3.3 evaluation of metronome

This section evaluate the autonomic control loop consisting of HRM, which carries on the observe phase, and Metronome that takes care of performing decisions and putting them in practice through actions. We are particularly interested in answering the following questions:

1. can the autonomic control loop established by HRM and Metronome constrain the performance measurements of instrumented applications so as to drive them towards their SLOs?

2. how good is the heuristic inside Metronome?

5 Since Metronome is implemented in kernel-space we are discouraged from using the floating point unit to perform our computations; thus, we perform all the multiplication before doing any division.

3.3.1 *Evaluation Platforms and Configurations*

We evaluated Metronome on a workstation with an Intel Core i7-870 Processor and 4 GB of DDR3-1066. The processor features 4 cores operating at 2.97 GHz and sharing 8 MB of Last-Level Cache (LLC). We disabled the Enhanced Intel SpeedStep Technology and the Intel Turbo Boost Technology to prevent the processor entering P-states with clock frequency lower or higher than the nominal when a subset of the cores is executing. We disabled the Intel Hyper-Threading Technology (HTT) to simplify our analysis.

We configured Debian 6.0 to boot with the Linux kernel 2.6.35 enhanced with HRM. HRM was configured to support at most 64 tasks per group (*i.e.,* one memory page) and a depth of the variable-term performance measurement of up to 1024. High resolution timers were scheduled to aggregate the per-task counters every 100 ms.

We assessed the behavior of the autonomic control loop made up of HRM and Metronome through a subset of the PARSEC 2.1 benchmark suite [50, 42], which provides a set of representative real-world applications with their native inputs. We ran multi-program workloads with multi-thread applications so as to saturate the evaluation platform and create contention, which is the scenario Metronome tries to address.

3.3.2 *Meeting SLO within Multi-Program Workloads*

We constructed 3 workloads of 2 applications from a set of 5 instrumented applications; Table 6 reports the details.[6]

We ran each multi-program workload with CFS and the combination of HRM and Metronome 100 times to achieve statistical relevance. We used the performance measurements collected when running with CFS to set achievable

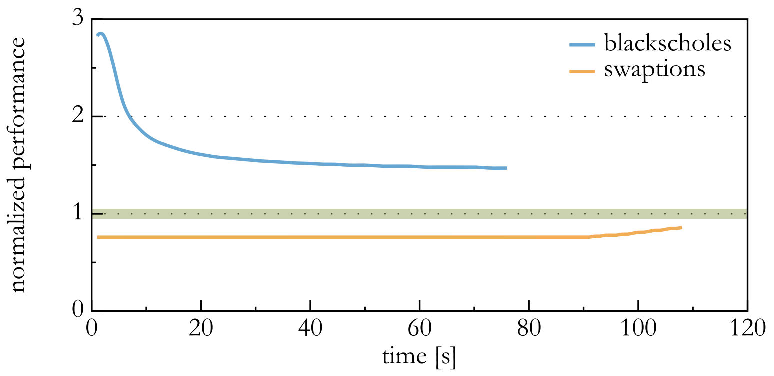
6 *ferret* runs with 16 threads of execution since it employs a pipeline with 4 stages and a pool of 4 threads per stage.

Table 6.: Details of the multi-program workloads consisting of multi-thread applications taken from the PARSEC 2.1 benchmark suite. The multi-thread applications execute with the options `-i native -n 4`

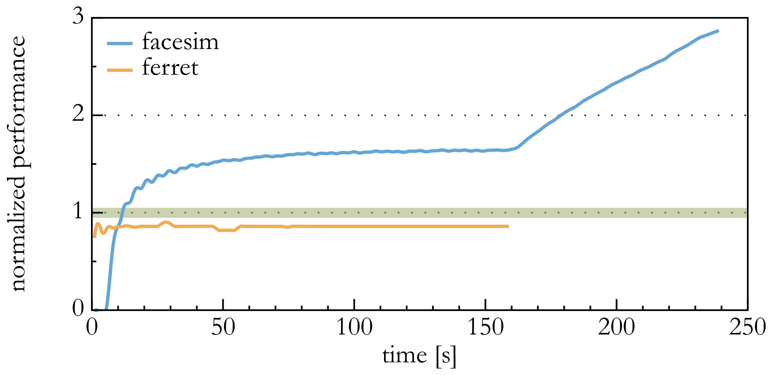
workload	application	threads
A	*blackscholes*	4
	swaptions	4
B	*facesim*	4
	ferret	16
C	*facesim*	4
	fluidanimate	4

SLOs with HRM when running with Metronome. Table 7 reports arithmetic averages (*i.e.,* avg) and standard deviations (*i.e.,* std) over the repetition divided by workload, application, and scheduling solution. As expected, the overall execution time of the multi-program workloads are almost identical when executing with CFS or Metronome. In fact, the total number of instructions to execute does not change and the evaluation platform is always saturated, even when one of the two multi-thread applications within the multi-program workloads end the execution before the other since Metronome cannot act in absence of contention. We reported a slightly higher standard deviation for Metronome; we believe this is due to the heuristic it employs.

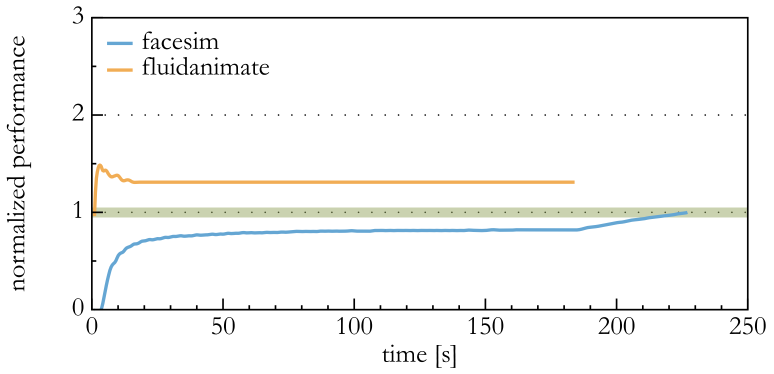
Figures 20 and 21 shows 6 out of the 600 executions of the multi-program workloads A, B, and C with CFS (see Figures 20a to 20c) and with HRM & Metronome (see Figures 21a to 21c). The green areas represent the SLOs, which obviously CFS cannot meet since it partitions fairly the CPU bandwidth across the multi-thread applications of the multi-program workloads. On the other hand, HRM & Metronome drive the multi-thread applications towards their SLOs. Figure 21b clearly shows the work-conserving nature of Metronome, which assigns the whole CPU bandwidth whenever there is no contention it, something that happens as soon as *ferret* ends its execution (before *facesim*).



(a) Multi-program workload A

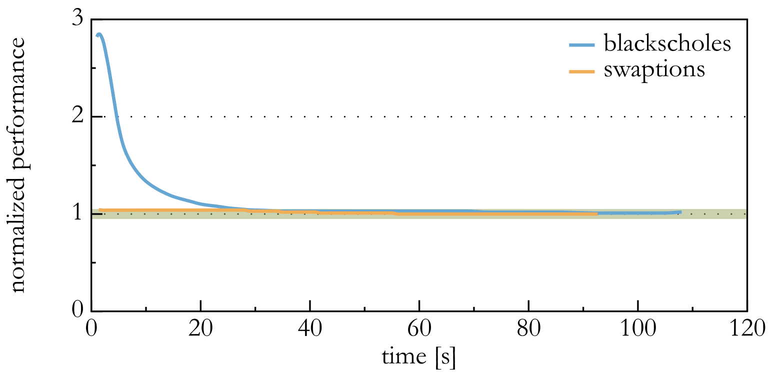


(b) Multi-program workload B

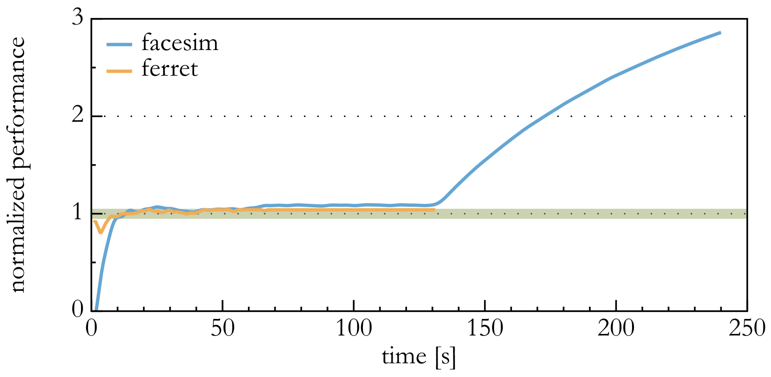


(c) Multi-program workload B

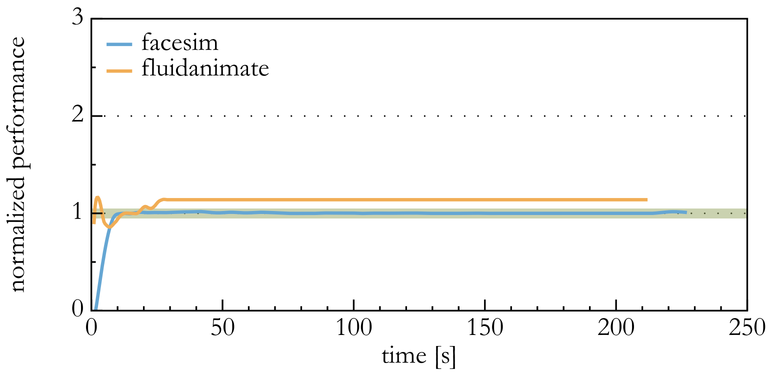
Figure 20.: Sample executions of multi-program workloads A, B, and C with CFS.



(a) Multi-program workload A



(b) Multi-program workload B



(c) Multi-program workload C

Figure 21.: Sample executions of multi-program workloads A, B, and C with HRM & Metronome.

Table 7.: Averages and standard deviations over the execution times of multi-thread applications within multi-program workloads. Each multi-program workload is executed 100 times with either CFS or HRM & Metronome

workload	application	CFS		HRM & Metronome	
		avg [s]	std [s]	avg [s]	std [s]
A	*blackscholes*	76.718	0.406	109.020	0.577
	swaptions	108.504	0.284	93.796	0.428
B	*facesim*	239.982	0.561	241.311	0.696
	ferret	159.776	0.426	132.043	0.567
C	*facesim*	227.968	0.588	228.240	0.742
	fluidanimate	184.936	0.560	212.991	0.562

We demonstrated the ability of the autonomic control loop established by HRM and Metronome to drive multi-thread applications towards their SLOs. We show how good is the heuristic inside Metronome by computing the mean absolute error through Equation (7).

$$\epsilon = \frac{\sum_{o}^{n} |\bar{r} - r|}{n} \tag{7}$$

Where o is the o-th observation of the performance measurement r, which is the throughput of the multi-thread application and n is the number of observations. $\bar{r}$ represents the SLO of the multi-thread application. The computation of ϵ accounts for all the observations collected while there is contention for the CPU bandwidth (*i.e.,* the two multi-thread applications were still executing concurrently). ϵ goes from 0.06 % for *swaptions*, which is an extremely regular applications that does not suffer when sharing resources (*e.g.,* the shared LLC) [62], to 14.15 % for *blackscholes* that takes a lot of time to converge to the SLO. However, most of of the multi-thread applications present values of ϵ below 2 %.

3.4 DESIGN AND DEVELOPMENT OF METRONOME++

Section 3.2 focused on the problem of assigning CPU bandwidth using a fair method that considers the performance measurements collected through HRM and the user-stated SLOs. The problem of distributing CPU bandwidth among multiple applications was born back in the days with multi-tasking operating systems and it is still an interesting problem because of the trade-off among the different policies we mentioned earlier. However, with the rise of multiprocessor systems, multicore processor systems, and, as a consequence, of parallel and embarrassingly parallel applications, another objective other than fairly distribute CPU bandwidth grew up in importance: balance the assignment of tasks to CPU cores.

Metronome (formerly known as Performance-Aware Processor Allocator ($(PA)^2$) [63]) couples a resource allocation policy and a mechanism to build a sort of second-level scheduling solution on top of the scheduling infrastructure of the Linux kernel. The idea behind Metronome++ resemble that behind Metronome, re-define the concept of balance to support applications (*i.e.,* groups of tasks, see Chapter 2) that are bound to user-stated SLOs. The additional level of scheduling adjusts the concept of balance and introduces the desirable property of predictability.

3.4.1 *Modeling the Behavior of Parallel Applications*

With Metronome we explored a simple yet effective solution employing a heuristic to affect CPU bandwidth distribution across multi-thread applications within multi-program workloads. Metronome++ follows a different route and tries to exploit applications characteristics to assign CPU cores to multi-thread applications running either solo or within multi-program workloads.

Metronome++ exploits a second-order polynomial to estimate the relationship between the performance measurements collected through HRM and the resource allocations. The use of a second-order polynomial is justified by the behavior of the speedup curve of parallel applications, which usually grows less than linearly when the number of allocated CPU cores grows linearly [58, 9].

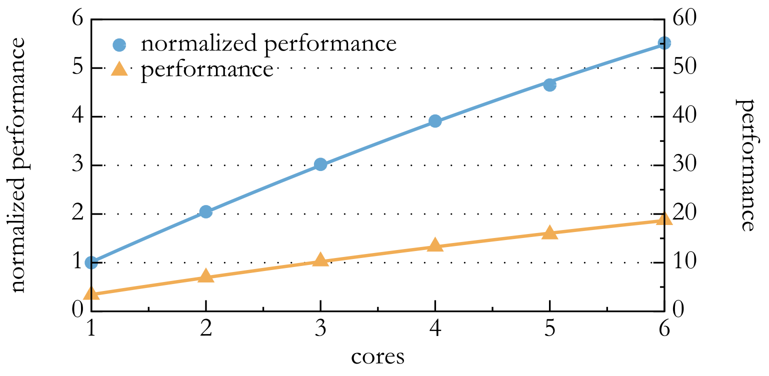
One way of modeling a computing system consists of determining a function f through an accurate mathematical analysis of the operations carried out by the system. Unfortunately, analytical methods frequently fail due to the complexity of modern systems. A more practical way of modeling a computing system makes use of an estimate of the computing system behavior obtained through statistical analysis. The model consists of a function f with a finite number of parameters and independent variables. Apart from choosing the function f, its parameters need to be identified; this operation can be realized through algorithms such as least squares or its variants.

Equation (8) reports the second order polynomial Metronome++ exploits.

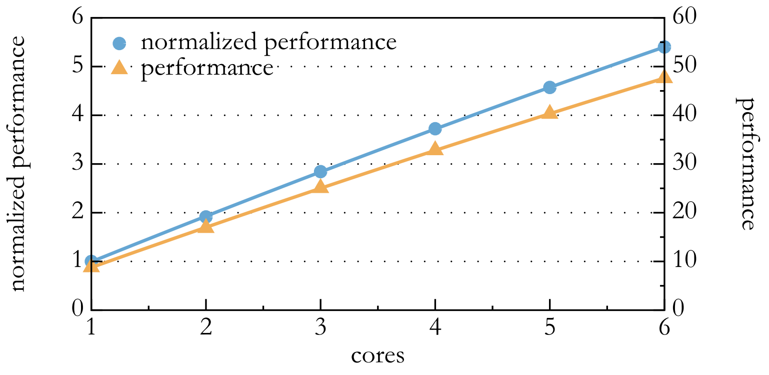
$$s_g(k) = p_1 \cdot c_g(k)^2 + p_2 \cdot c_g(k) + p_3 \quad \text{s.t.} \quad p_1 < 0 \tag{8}$$

Where $c(k)$ is the number of allocated CPU cores to the g-th group between the sampling intervals k and $k-1$, $s_g(k)$ is the reference performance measurement for the g-th group normalized to the reference performance measurement with a singlecore (*i.e.,* the speedup), and p_1, p_2, p_3 are unknown best-fit parameters which are to be estimated. Figure 22 explains why we must use the speedup instead of the raw performance measurements to estimate the best-fit parameters of the second order polynomial. In fact, the speedup remains almost stable across the execution phases of parallel applications, like it happens for the *x264* video encoder from the PARSEC 2.1 benchmark suite [50, 42], while the raw performance measurements vary.

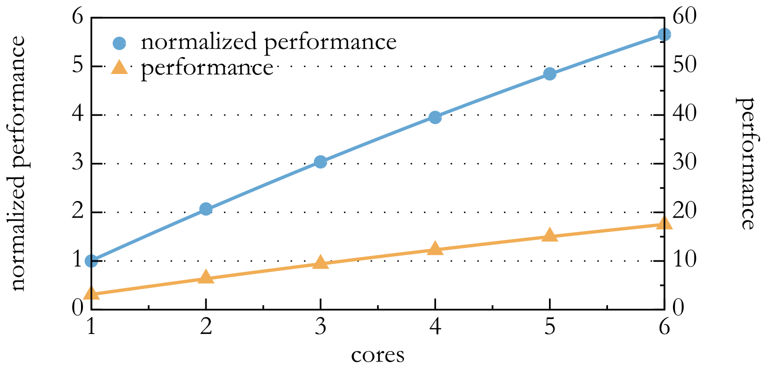
Within Metronome the reference performance measurement accounted for the whole execution of applications. Conversely, Metronome++ requires the reference performance measurement to account for a time window so as to better shape the relationship between performance measurements and resource allocation. When applications start, Metronome++ enters an initial



(a) Applications characteristics for *x264* phase 1.



(b) Applications characteristics for *x264* phase 2.



(c) Applications characteristics for *x264* phase 3.

Figure 22.: Comparison between normalize performance (*i.e.*, speedup) and performance across the execution phases of *x264*.

exploration phase in which applications are assigned with a progressively increasing number CPU cores so as to better understand their speedup curves. The observations collected during the exploration phase constitute the first set of tuples to apply the least squares algorithm and estimate the best-fit parameters, which will change across the whole execution whenever the number of applications grows or shrinks.

3.4.2 *Modeling the Changes between Phases*

Applications can show widely different behavior over their execution, and these behaviors can be seen from the smallest (*i.e.,* over few thousands of instructions) to the largest (*i.e.,* over the whole execution of the application) scale. Recurring program behaviors are commonly known as execution phases [64]. On one hand, execution phases may be due to an alternation of a CPU- and memory-intensive streams of instructions. On the other hand, they may be due to the varying complexity of the program input.

As an example, consider once again the *x264* video encoder. It operates on macro-blocks of pixels which have the fixed size of 16×16 pixels. Various techniques are used to detect and eliminate data redundancy, of which the most important is motion compensation, employed to exploit temporal redundancy between successive frames. This is usually the most expensive operation to be executed for encoding a frame, with a very high impact on both the final compression ratio and the time spent encoding each frame (*i.e.,* high variance among different frames). The compressed output frames can be encoded in one of three possible ways [42]:

- i-frame, which includes the entire image and does not depend on other frames;
- p-frame, which includes only the changed parts of an image from the previous i or p-frame;

- b-frame, which is constructed using data from the previous and next i or p-frames.

The amount of time needed to encode each frame depends on its type and frame types alternate within a video. Alternation happens depending on the program input (*i.e.,* the video for *x264*) and causes *x264* to yield radically different performance measurements with different program inputs. Because of this reason we identified the need to model the changes between execution phases to identify the workload complexity and consistently yield the user-stated SLOs.

Within Metronome++ we employ an Exponential Moving Average (EWA) adaptive filter. We initially evaluated both a Moving Average (MA) and an EWA to construct the adaptive filter. The former tends to behave poorly with rapidly varying execution phases, unless the number of samples used is small (*e.g.,* 2 or 3). However, employing a small number of samples may make the adaptive filter more subject to noise. The EWA behaves better with rapidly varying workloads even with a larger number of samples, which helps cancel occasional noise. Within Metronome++ we compute a workload predictor by applying the EWA adaptive filter to the speedup over the reference performance measurement (*i.e.,* the same value employed to estimate the best-fit parameters, see Section 3.4.1 divided by the number of allocated CPU cores as reported in Equation (9).

$$p = \sum_k w_k \frac{s_g(k)}{c_g(k)} \tag{9}$$

$$\sum_k w_k = \sum_{i=1}^{n} a^{-i} = 1 \tag{10}$$

Where p is the workload predictor, w_k are the weights of the EWA adaptive filter whose summation converges to 1,[7] a is parameter whose value depends on the depth of the EWA adaptive filter, $s_g(k)$ is the speedup over the reference

7 Let us consider an EWA adaptive filter over the last 4 sampling intervals, which means k goes from 3 to 0, then $w_3 = a^{-1}$, $w_2 = a^{-2}$, $w_1 = a^{-3}$, and $w_0 = a^{-4}$.

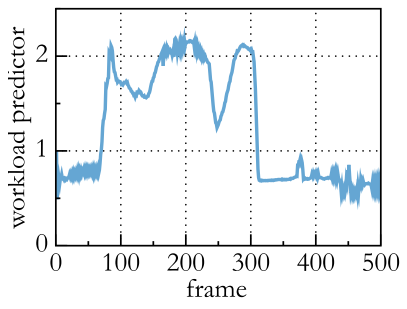
performance measurement for the g-th group, and $c_g(k)$ is the number of allocated CPU cores for the g-th group between the k and $(k-1)$-th sampling intervals.

Figure 23 shows the behavior of the workload predictor for *x264* running with different numbers of threads, from 2 to 6. The curves are consistent across the different scenarios, thus making the proposed workload predictor a suitable candidate to analyze the transition among execution phases and aid the resource allocation policy.

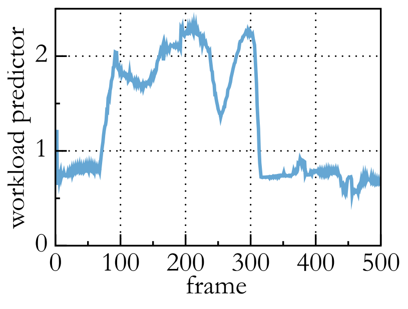
3.4.3 *Details*

The design of Metronome++ requires using floating point instruction to perform the estimation of the best-fit parameters of the second order polynomial and the computation of the workload predictor. Due to this reason, we implement Metronome++ by means of a split solution with a user-space daemon per group, which exploits the user-space consumer functionality of HRM and *libhrm*, that carries on the operation requiring floating point instructions and a kernel-space daemon per group that is a kernel task whose job is moving tasks among the runqueues of the scheduling infrastructure of the Linux kernel following the decision of the user-space daemon. The user- and kernel-space daemons communicates over memory pages shared across the boundary between kernel- and user-space (*i.e.,* the same way producers and consumers do with HRM).

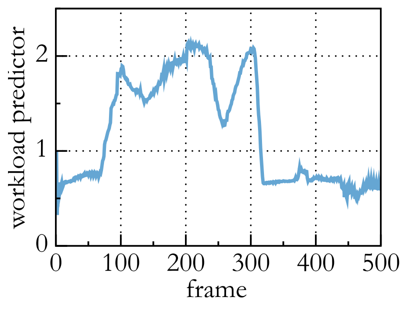
The kernel-space daemons performs fractional mappings of tasks to CPU cores by assigning the higher number of CPU cores for a portion of the sampling interval and the lower number of CPU cores for the remaining. For example, if the sampling interval is $1\,s$ and the outcome of the user-space daemon is 2.4 CPU cores, the kernel-space daemon allocates 3 of them for $0.4\,s$ and 2 of them for $0.6\,s$. In addition, the kernel-space daemons strive to minimize the number of moves of tasks among runqueues to favor locality, even though with CMPs this is less of an advantage than with multi-socket solutions.



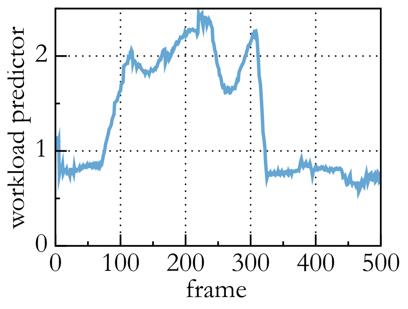
(a) *x264*, 2 threads.



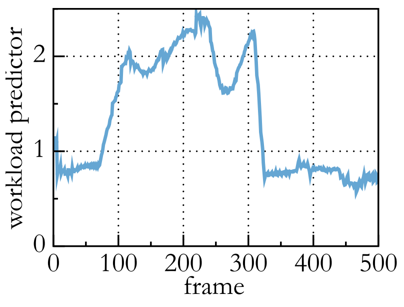
(b) *x264*, 3 threads.



(c) *x264*, 4 threads.



(d) *x264*, 5 threads.



(e) *x264*, 6 threads.

Figure 23.: Behavior of the workload predictor for *x264* running with different numbers of threads, from 2 to 6. Values are consistent across the different scenarios making the workload predictor a suitable choice to analyze the transition among execution phases.

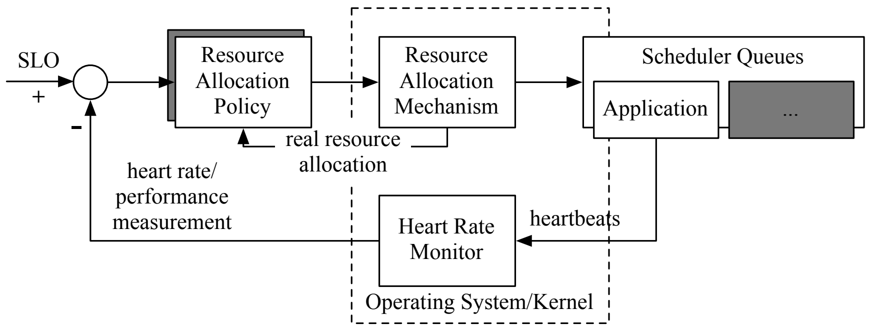


Figure 24.: Exemplified system architecture of Metronome++. Each application (*i.e.,* ensemble of tasks) attaches to HRM; the resource allocation policy decides whether the number of CPU cores is enough through the application model and workload predictor, eventually instructing the resource allocation mechanism to give the application more or less CPU cores.

Figure 24 depicts the overall system architecture of Metronome++, identifying which components reside in kernel-space (inside the dashed line) and user-space (outside the dashed line).

3.5 EVALUATION OF METRONOME++

This section evaluate the autonomic control loop consisting of HRM, which carries on the observe phase, and Metronome++ that takes care of performing decisions through the user-space daemons and putting them in practice through the kernel-space daemons. We are particularly interested in answering the following questions:

1. what is the sensitivity of Metronome++ with respect to the sampling interval, the number of samples employed to fit the second order polynomial and to the number of samples used to compute the workload predictor?

2. can the autonomic control loop established by HRM and Metronome++ constrain the performance measurements of instrumented applications so as to drive them towards their SLOs?
3. can this goal be attained for both single- and multi-program workloads?

3.5.1 *Evaluation Platforms and Configurations*

We evaluated Metronome++ on a Dell Precision T3500 workstation with an Intel Xeon W3530 Processor and 12 GB of DDR3-1066. The processor features 4 cores operating at 2.80 GHz and sharing 8 MB of LLC. We disabled the Enhanced Intel SpeedStep Technology and the Intel Turbo Boost Technology to prevent the processor entering P-states with clock frequency lower or higher than the nominal when a subset of the cores is executing. We disabled the Intel Hyper-Threading Technology (HTT) to simplify our analysis.

We configured Debian 7.0 to boot with the Linux kernel 3.3 enhanced with HRM and the kernel-space partition of Metronome. HRM was configured to support at most 64 tasks per group (*i.e.,* one memory page) and a depth of the variable-term performance measurement of up to 1024. High resolution timers were scheduled to aggregate the per-task counters every 100 ms.

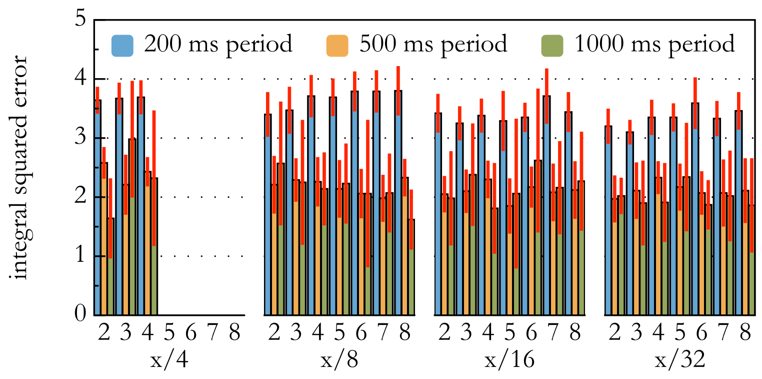
We assessed the behavior of the autonomic control loop made up of HRM and Metronome++ through a subset of the PARSEC 2.1 benchmark suite [50, 42], which provides a set of representative real-world applications with their native inputs. We ran both single- and multi-program workloads with multi-thread applications so as to saturate the evaluation platform.

3.5.2 *Sensitivity Analysis*

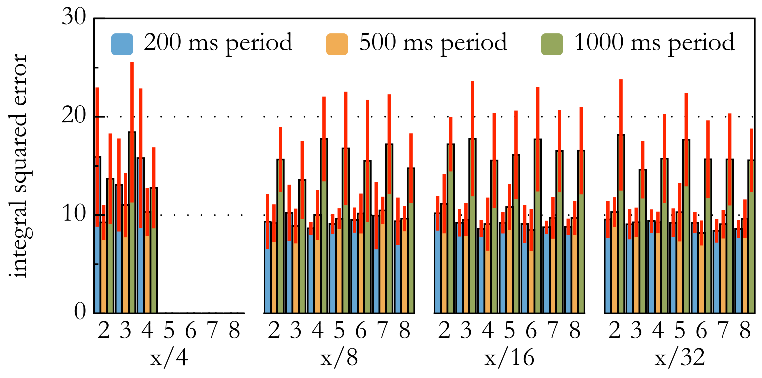
The aim of this section is studying the sensitivity of Metronome++ to the values of its parameters. The sampling interval is likely the most import parameter.

We performed experiments with a sampling interval of 200, 500, and 1000 ms. Smaller sampling interval may result in faster responses to the reference performance measurement variance but higher sensitivity to occasional noise. Conversely, longer sampling interval may result in slower to no response to the reference performance measurement variance but lower sensitivity to occasional noise. The number of samples to fit the second order polynomial shaping the relationship between the reference performance measurement and the number of allocated CPU cores and the number of samples to compute the workload predictor are the other knobs that may change the behavior of Metronome. We experimented with 4, 8, 16, and 32 samples for the former and 2, 3, 4, 5, 6, 7, and 8 samples for the latter. Figures 25a and 25b show the Integral Squared Error (ISE) on the long-term performance metric and on the variable-term performance metric for *x264*, respectively. We decided to employ *x264* for this experiment since it is the more challenging application within the PARSEC 2.1 benchmark suite considering our goal of achieving a user-stated SLO, which in this case was within 5 % of 10 frames/s. The error bars represent the standard deviations over 100 executions with the same configuration. As expected, the experimental results show that a short sampling interval (*i.e.,* 200 ms) is the best choice to fit the variable-term (say short-term) performance metric but, at the same time, a poor choice to fit the long-term performance metric. The exact opposite happens for a long sampling interval (*i.e.,* 1000 ms). Chosen the 500 ms sampling interval, the experimental results also show that Metronome++ is not very sensitive to the number of samples employed to fit the second order polynomial and to compute the workload predictor. Overall, a configuration with a sampling interval of 500 ms, 16 samples to fit the second order polynomial, and 5 samples to compute the workload predictor seems a reasonable choice.[8]

8 <500,16,5> is the configuration we used to carry one the remaining of the evaluation.



(a) Long-term performance metric.



(b) Variable-term (short-term) performance metric.

Figure 25.: Sensitivity of Metronome++ to its parameters. The x within the x-label is the number of samples to compute the workload predictor while value after "/" is the number of samples to fit the second order polynomial.

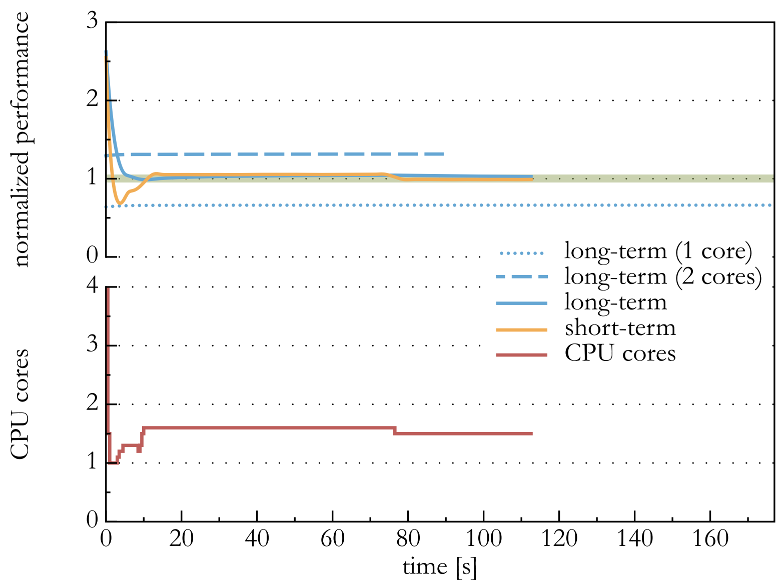


Figure 26.: *blackscholes*, 4 threads, SLO 8500000 options/s.

3.5.3 *Meeting SLO within Single-Program Workloads*

In this section, we show the dynamic of the performance measurement when applications execute on top of Metronome++. We show how different static resource allocations do not yield the user-state SLO while Metronome++ is capable of meeting them thanks to its modeling solutions. Figure 26 shows the behavior of Metronome++ executing *blackscholes*. *blackscholes* is a very regular application whose management is hardly a challenge for Metronome++; however, both the static resource allocations (*i.e.*, 1 and 2 CPU cores) cannot possibly satisfy the user-stated SLO. Figures 27 and 28 depicts the same behavior for both *canneal* and *swaptions*.

Figure 29 shows the dynamic of the performance measurement when dealing with *x264*. As already mentioned, *x264* is a complex application to manage, mainly because of its heavy dependence on the program input. This is likely

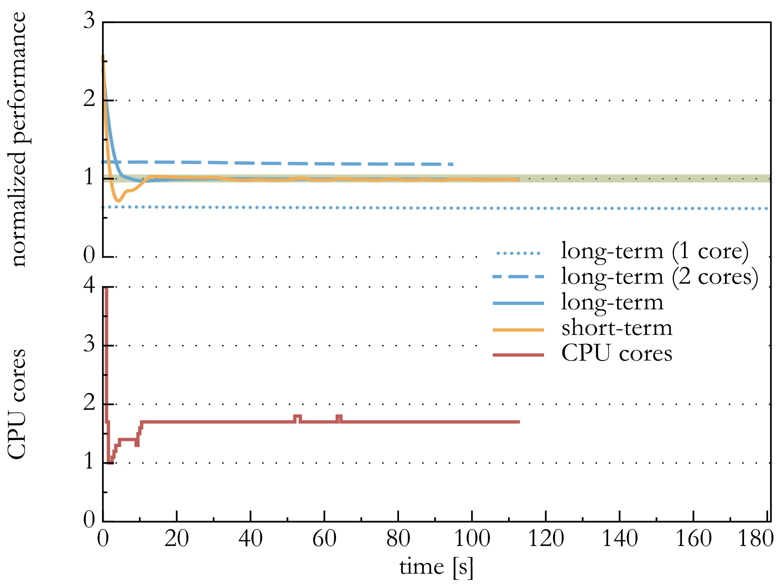


Figure 27.: *canneal*, 4 threads, SLO 800000 swaps/s.

the perfect application to evaluate Metronome. The resource allocation policy and mechanism are continuously forced to alter the number of CPU cores granted to *x264* in order to satisfy the user-stated SLO (blue squares).

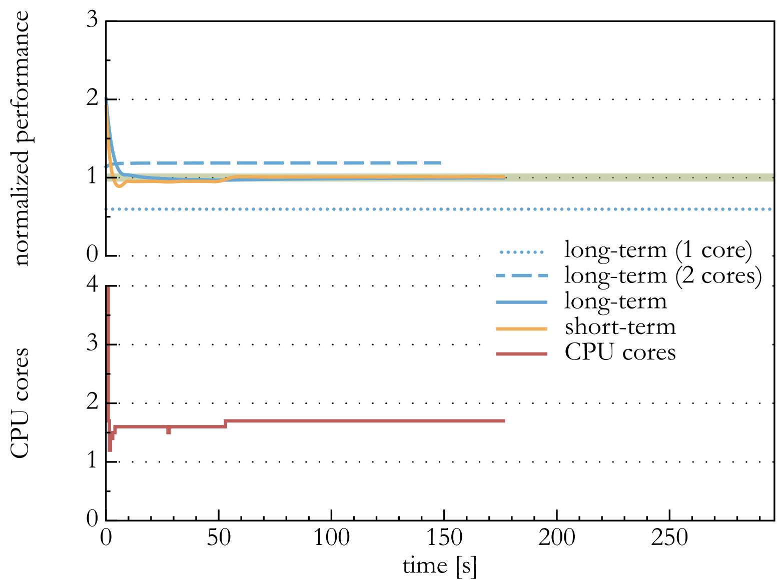


Figure 28.: *swaptions*, 4 threads, SLO 45000 Monte Carlo (MC) simulations/s.

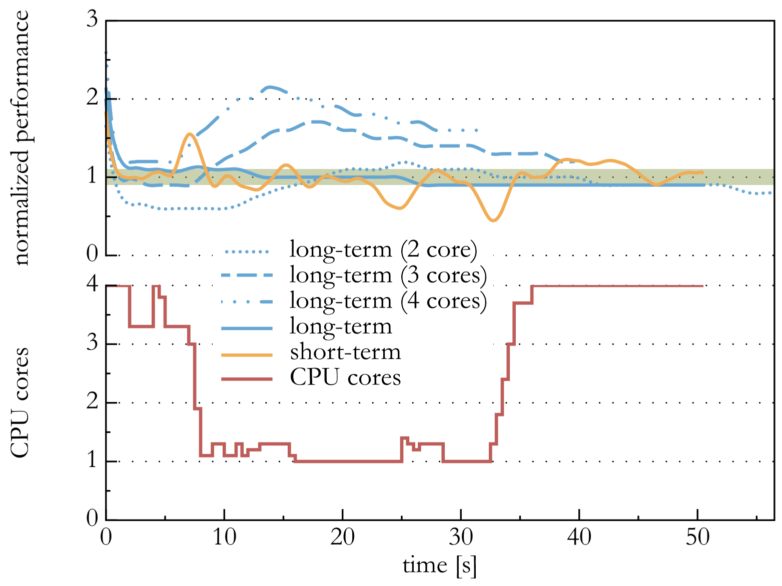


Figure 29.: *x264*, 4 threads, SLO 10 frames/s.

Table 8.: Details of the multi-program workloads consisting of multi-thread applications taken from the PARSEC 2.1 benchmark suite. The multi-thread applications execute with the options `-i native -n 4`

workload	application	threads
A	*blackscholes*	4
	canneal	4
B	*swaptions*	4
	swaptions	4
C	*x264*	4
	x264	4

3.5.4 *Meeting SLO within Multi-Program Workloads*

We constructed 3 workloads of 2 applications from a set of 4 instrumented applications; Table 8 reports the details. Figures 30 to 32 illustrate the behavior of Metronome++, which is consist over multiple executions, with the multi-program workloads when it is possible to meet the non-trivial[9] user-stated SLOs. In these figures, the y-axis report the online performance measurements normalized to the performance requirements (*i.e.,* the SLOs).

The behavior of Metronome++ remains very consistent between executing a single-program workload (see Figure 26) and a multi-program workload (see Figure 30): the dynamic of the performance measurement of *blackscholes* is very similar with the same SLO.

Figure 32 displays the behavior of Metronome++ with the most complex multi-program workload (C) consisting of two instances of *x264*. Resource allocation in this scenario is particularly difficult since both the instances of *x264* require the whole CMP during the execution. Reaching perfect isolation and predictability in this scenario is not possible when acting only on CPU core

9 We say that user-stated SLOs are non-trivial when there are no static resource allocation (*i.e.,* $<1,1>$, $<1,2>$, $<1,3>$, $<2,1>$, $<2,2>$, and $<3,1>$) such that the multi-thread applications within the multi-program workloads can meet their SLOs.

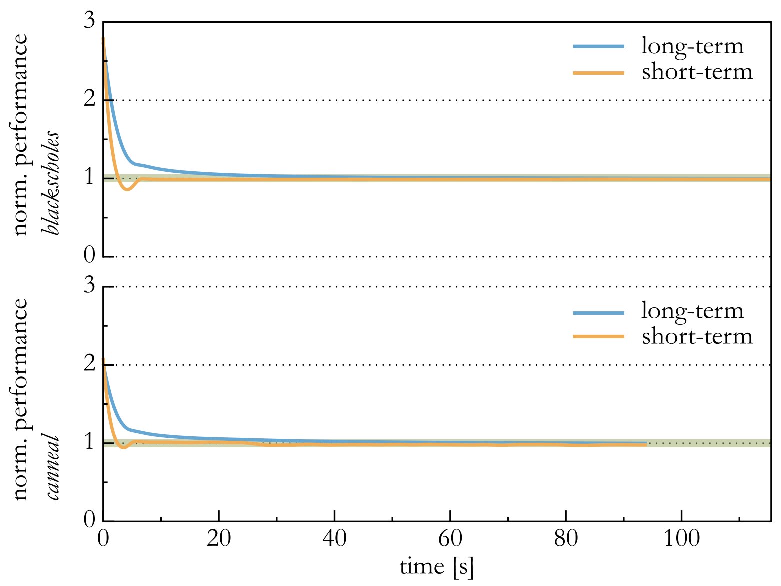


Figure 30.: Multi-program workload A. *blackscholes*, 4 threads, SLO 8500000 options/s. *canneal*, 4 threads, SLO 950000 swaps/s.

allocation due to the need to leverage the underlying time-sharing scheduling infrastructure. However, Metronome++ is capable of satisfying both the SLOs even if the variable-term performance measurements are visibly noisier than in the single-program workload (see Figure 29).

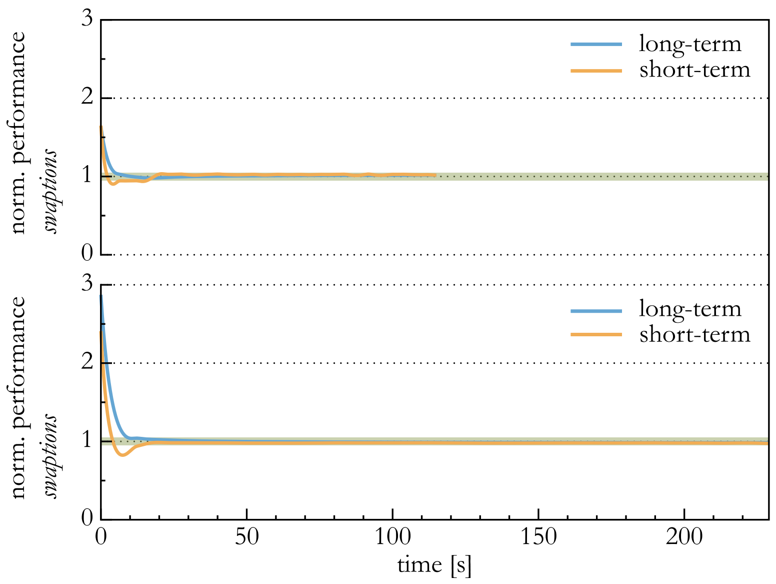


Figure 31.: Multi-program workload B. *swaptions*, 4 threads, SLO 65000 MC simulations/s. *swaptions*, 4 threads, SLO 35000 MC simulations/s.

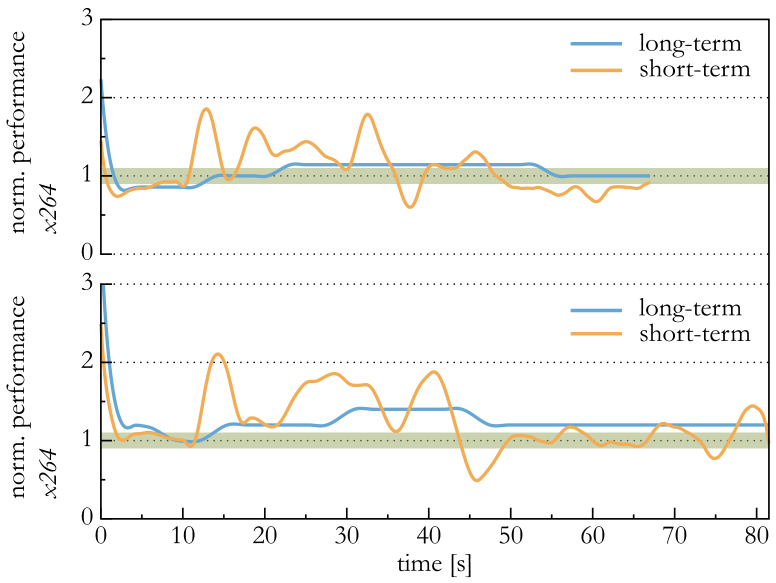


Figure 32.: Multi-program workload C. *x264*, 4 threads, SLO 9 frames/s. *x264*, 4 threads, SLO 7 frames/s.

3.6 RELATED WORK

In the latest years, there has been extensive research on strategies to maximize performance or performance under a power budget within multi-program workloads of multi-thread applications running on top of CMPs.

Many approaches focused on the partitioning of the cache hierarchy [65–69]. Others addressed the problem at the very end of the memory hierarchy altering the behavior of memory controllers [70–72]. Other focused on the problem of assigning tasks to CPU cores [73, 58, 6] or change the CPU bandwidth allocation [74]. Researchers also tackled the problem of maximizing performance possibly under a power cap with more comprehensive frameworks capable of managing more than one resource at the same time [75, 76]. Unfortunately, none of these solutions is focused on achieving user-stated SLOs.

Similarly to the work of Fedorova et al. [74], Metronome assigns CPU bandwidth to applications within multi-program workloads. However, Metronome focuses on achieving user-stated SLOs instead of maximizing performance, a scenario we believe we will be much more compelling in the future were resources will be abundant. Both the works (*i.e.,* Metronome and the related work) employ a heuristic resource allocation policy.

Metronome++ is closely related to the works proposed by Corbalán et al. [73], Sasaki et al. [58]; these two related work are quite similar since the both exploit the Amdahl's law to model the scalability characteristics of parallel applications and redistribute the CPU cores accordingly. The work by Sasaki et al. [58] also accounts for the different execution phases applications may go through and changes the allocations accordingly. Metronome++ makes use of a second order polynomial to model the scalability characteristics of parallel applications and decides how to distribute CPU cores accordingly. However, it does so with the objective of achieving user-stated SLOs; objectives expressed through high-level performance metrics that are meaningful to both applications' developers and users. Conversely, *SBMP* [58] exploits Instructions Per Cycle (IPC), which is hardly meaningful and may even be the wrong choice for parallel applications [33]. In addition, Metronome++ supports execution phases through a workload predictor that proved effective in understanding the dynamic behavior of applications both when executing solo and within multi-program workloads. Each of these works employ a model-based resource allocation policy.

There exists also more comprehensive frameworks in terms of the resources they manage or the SLO they can achieve. *METE* [77] is one of those; it exploits sound control theory to drive the partitioning of CPU cores, the cache hierarchy, and the Dynamic Random-Access Memory (DRAM) bandwidth. *METE* accepts user-stated SLOs expressed through low-level performance metrics (*i.e.,* IPC) and it is unclear how the "filter" they propose can map those metrics to high-level ones since the latter may not be representative of the former [33]. In addition, the authors simulated the behavior of *METE* since the cache hierarchy and the DRAM bandwidth cannot be partitioned in commodity CMPs.

SEEC [78, 53, 79] is the second closely related comprehensive solution, which also supports an experimental processor. *SEEC* exploits sound control theory and machine learning to drive both CPU core allocation, DVFS, and application-level adaptation. Neither Metronome nor Metronome++ cannot directly compare to *SEEC* since the latter accepts user-stated SLOs bounding both a high-level performance metric [21] and power consumption. However, both Metronome and Metronome++ are competitive with the CPU core allocation policy proposed with *SEEC*.

Orthogonal approaches [80, 81] dynamically adjust the number of threads within parallel applications to optimize the overall efficiency of computing systems.

3.7 SUMMARY

Metronome and Metronome++ proved effective in managing CPU bandwidth and cores, respectively, when closing the autonomic control loop with HRM. Both the solutions proved effective when dealing with the execution of multi-program workloads consisting of multi-thread applications. In addition, Metronome++ can handle single-program workloads so as to minimize the amount of resource allocated, thus opening to adaptive power management solutions. Both the heuristic resource allocation policy within Metronome and the model-based resource allocation policy within Metronome++ yields high accuracy (see the mean absolute errors for the former and the ISE for the latter).

This chapter presented various contributions from published work [8, 9].

As future work we intend to integrate the adaptive performance management solutions presented so far with the Dynamic Thermal Management (DTM) solutions we will present later in this dissertation (see Chapter 4) that we already discussed in our publications [12, 9, 11].

4 ADAPTIVE THERMAL MANAGEMENT

Constraining power and temperature has become a dominant aspect of the design of both processors and computing systems. The supply voltage decrease has lost its pace even though the feature size is shrinking constantly. This phenomenon, which is the result of Dennard's scaling law failure, coupled with the increasing number of transistors per unit of area, which is due to Moore's law, translates into a growing power density. Power density is one of the root of high temperature, which impairs performance, energy consumption, and reliability. System researchers started investigating Dynamic Thermal Management (DTM) techniques to address the trade-off between performance and temperature. Hardware DTM can effectively constraint the temperature of processors, eventually shutting them, thus guaranteeing safety. At the same time, hardware solutions can negatively affect established Service-Level Agreements (SLAs) and Service-Level Objectives (SLOs). On the other hand, software solutions rely on hardware for safety, but does not indiscriminately trade-off performance for temperature. We propose ThermOS, an extension for commodity operating systems that harnesses formal feedback control and idle cycle injection to decrease thermal emergencies while showing better efficiency than commodity and cutting edge techniques.

4.1 INTRODUCTION

The shift from singlecore superscalar processors to multicore processors was a tentative response to address the inability of respecting Joy's law: the peak computer speed doubles each year [2]. If parallel software is available, a multicore processor made up of an ensemble of smaller cores [82], which harness thread-level parallelism, can outperform a massive singlecore superscalar processor exploiting instruction-level parallelism within the same power budget.

In the last decade we assisted to the proliferation of multicore processors such as Chip-Multiprocessors (CMPs) and Multiprocessor System-on-Chips (MPSoCs) characterized by a constantly increasing number of transistors made possible by the ever-decreasing feature size [3]. However, recent lithographic technologies do not abide Dennard's scaling law [1] causing power density of a multicore processors to approach that of a nuclear reactor. Power density increases as the scaling of clock frequency and number of transistors outpaces the downscaling of supply voltage. The consequent rise of temperature due to the inability of packages to dissipate heat heavily influences the design of both processors and computing systems.

Maintaining temperature under control is crucial for performance, energy consumption, and reliability of integrated circuits: a higher temperature increases leakage current and leads to a sharp increase of energy consumption [83] and to drastic decreases of both throughput [84] and Mean Time To Failure (MTTF) [85]. Researchers from the computer architecture, compiler, and operating system communities put efforts in addressing this issue. Our work pursues the same objective.

We propose ThermOS (Thermal Operating System) [11], an extension for commodity operating systems, which provides DTM through formal feedback control and idle cycle injection [17] for multi-programmed workloads. ThermOS specifically targets commodity CMPs, which cannot benefit from the latest architectural and micro-architectural advancements. However, we believe that ThermOS could benefit even further from both the architectural and micro-architectural evolution.

This chapter makes the following contributions:

- Propose and validate a linear discrete-time thermal model that describes the temperature behavior around the threshold when employing idle cycle injection for DTM;
- Derive a proportional-integral controller to drive idle cycle injection, demonstrate its asymptotic stability and robustness;
- Evaluate ThermOS on a commodity CMP with representative benchmarks showing its capabilities of managing multi-programmed workloads and addressing the trade-off between temperature and performance.

The remainder of this chapter is organized as follows. Section 4.2 makes a first high-level comparison between Dynamic Voltage and Frequency Scaling (DVFS) and ThermOS. Section 4.3 describes the linear discrete-time thermal model that enables ThermOS while Section 4.4 reports implementation details regarding each ThermOS component. Section 4.5 provides evidence that ThermOS achieves its goals. Section 4.6 surveys, at the best of our knowledge, related work and highlights benefits and drawbacks of ThermOS with respect to the state of art. Finally, Section 4.7 concludes this chapter.

4.2 dynamic thermal management: the bad and the good

Typical scheduling algorithms implement the race to idle approach: applications run as fast as possible to allow processors entering low power states as soon as possible. This behavior leverages the capability of decreasing energy consumption when employing low power states and delivers the best performance. Race to idle favors energy efficiency [86] and is beneficial for desktops, laptops, and mobiles, where interactive, low-utilization applications are common.

Conversely, race to idle leads to high temperature in servers and large-scale computing systems where non-interactive high-utilization applications prevail, incurring in additional costs to power Computer Room Air Conditioning

(CRAC) and Heat, Ventilation, and Air Conditioning (HVAC). CRAC and HVAC are in place to avoid exceeding the temperature threshold[1] and limit the number of DTM events that degrade vital measures such as throughput, latency, and missed deadlines.

4.2.1 *Dynamic Voltage and Frequency Scaling*

Researchers from the computer architecture community demonstrated the energy efficiency of per-core P-states [87] through DVFS in CMPs [88]. Each core within a CMP heats up differently depending on the manufacturing variability of the silicon, the floorplan, the application it is running, etc. [83] making the adoption of per-core P-states desirable to address temperature issues. However, providing such a fine-grained control in CMPs with more than few cores is uncommon [57] and most of the manufacturers ditch fine-grained control in favor of coarse-grained control.[2]

This scenario is especially harmful for multi-tenant environments. Each user is assigned a certain amount of resources and expects predictable performance. The occurrence of DTM events (*e.g.,* the activation of DVFS as a consequence of the overheating of a single core running a particularly Central Processing Unit (CPU)-intensive application) affects the whole CMP and impairs the performance of all the applications within a multi-programmed workload, regardless of the owner.

Figure 33 depicts this setting. We run *swaptions*, a CPU-intensive application from the PARSEC 2.1 benchmark suite [50, 42], and *apachebench*, an Input/Output (I/O)-intensive application, on two different cores. When running at the highest clock frequency, the core executing *swaptions* overheats (see the dashed red line in Figure 33), breaking the temperature threshold, while the core executing *apachebench* does not (see the dashed green line in Figure 33). When

1 The temperature threshold can be either a manufacturer-defined safety limit or an administrator-defined cap to lower the total cost of ownership.

2 Commodity CMPs support per-core P-states; unfortunately, this setting becomes effective only when cores operate in different C-states.

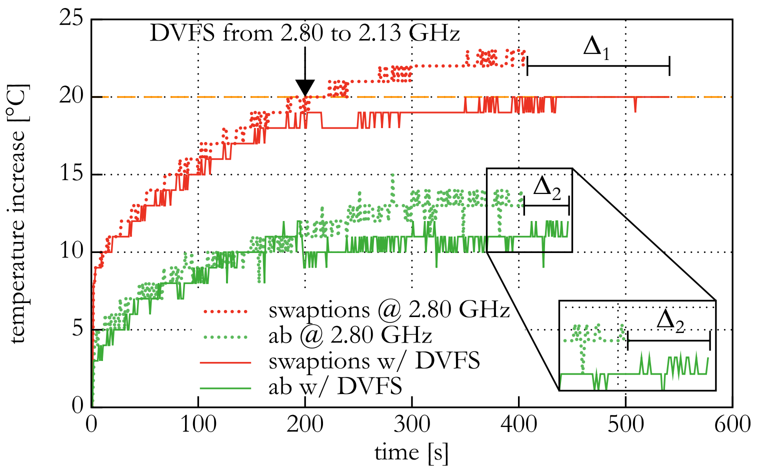


Figure 33.: Executions of a multi-programmed workload on a commodity CMP. DVFS decreases temperature with a chip-wide impact on performance.

the same multi-programmed workload executes on a CMP using chip-wide DVFS for DTM, each core is subject to the same supply voltage and clock frequency setting. The core executing *swaptions* does not overheat when DVFS is in place since it decreases the supply voltage and the clock frequency (see the point at 200 s in the execution in Figure 33); this directly translates into a run time increase (see red line and Δ_1 in Figure 33). Unfortunately, the same happens—on a smaller scale due to the I/O-intensive nature of the application—to *apachebench*, which does not cause overheating at the highest clock frequency (see the light green line). Hence, *apachebench* is unnecessarily slowed down (see green line and Δ_2 in Figure 33).

4.2.2 *Idle Cycle Injection*

The system-wide performance degradation of chip-wide DVFS is its main drawback. ThermOS addresses this issue by harnessing formal feedback control and idle cycle injection [17]. Let us reconsider the previous multi-tenant

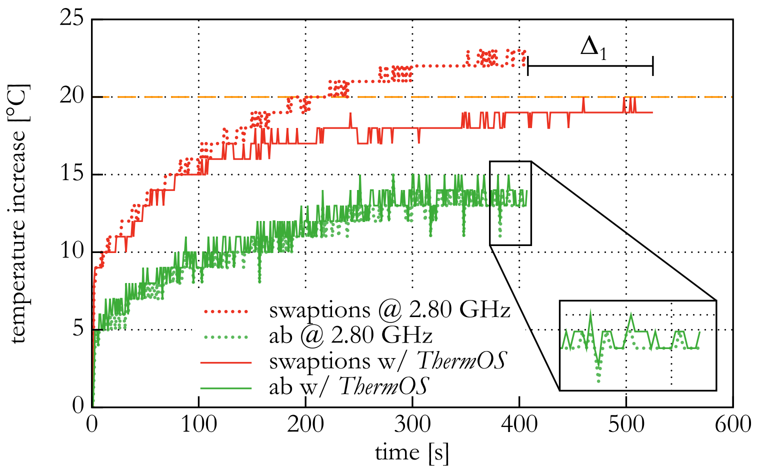


Figure 34.: Executions of a multi-programmed workload on a commodity CMP. ThermOS decreases temperature with a core-wide impact on performance.

scenario. ThermOS selectively throttles the execution of those applications whose cores are overheating without affecting the remaining cores and thus avoiding system-wide performance degradation. Figure 34 depicts this setting. ThermOS prevents the core running *swaptions* from overheating by increasing its run time (see red line and Δ_1 in Figure 34). At the same time, it does not impact the execution of *apachebench* (see green lines in Figure 34).

Almost all processors today support a handful of C-states—five on Intel "Ivy Bridge"—and this trend is spreading as processors dynamic power and thermal management gain momentum [89]. For example, our evaluation platform features an Intel Xeon Processor W3530 supporting the C0, C1/C1E, C3, and C6 states at the individual thread, core, and package level. A convenient interface accessible through the `MWAIT` instruction allows the operating system requesting low power states [90]. ThermOS exploits this interface to selectively throttle applications, thus decreasing temperature.

Flexibility makes idle cycle injection very interesting. For example, Google is already exploiting idle cycle injection through *kidled* [91] in some of its data centers while Intel recently merged *PowerClamp* [92]—a thermal driver that harnesses idle cycle injection—with the Linux kernel 3.9 (released April 29, 2013). Other approaches, like *Dimetrodon* [17], rely on idle cycle injection to provide preventive thermal management via probabilistic injection of idle time. We thoroughly compares *Dimetrodon* with ThermOS in Section 4.5.

4.3 thermal model

Electronic systems such as integrated circuits abide by a number of physical laws, some of which regulate the way temperature behaves, that can be expressed as mathematical relationships. These mathematical relationships are commonly referred to as first-principle models.

DTM can benefit from the adoption of first-principle models for accurate thermal modeling and many proposals can be found in the computer architecture literature. Brooks and Martonosi [93] model the temperature behavior through power consumption in the *Wattch* power analysis framework. Unfortunately, chip-wide power consumption is a poor proxy for temperature [83].

Skadron et al. [83] model the temperature behavior through a compact thermal model in the *HotSpot* thermal analysis framework. This solution is fairly accurate and explains the complete temperature behavior. The main disadvantage of the compact thermal model is the need for a considerable amount of micro-architectural details such as the floorplan of the functional units. This information may be available for obsolete designs but can only be guessed for current ones.

Zhou et al. [94] harnesses the compact thermal model to deploy a thermal-aware scheduler, which requires the complete temperature behavior since its objective is minimizing the temperature without hurting performance. However, the compact thermal model is simplified to make its adoption viable outside of a simulation environment and inside the Linux kernel.

Commodity designs cannot benefit from the latest advancements in micro-architectural DTM [95]. They rely on conservative techniques like DVFS to guarantee safety. Given safety for granted, portable software DTM techniques like idle cycle injection become attractive to address the trade-off between temperature and performance, as shown in Section 4.2.

4.3.1 *Thermal Model for Dynamic Thermal Management*

Deriving a thermal model that is meaningful for the whole range of commodity designs and DTM techniques is impractical. Because of this reason, we start simple with a linear discrete-time first-order model, which already proved a valuable approximation in many situations such as modelling the throughput [10] of applications belonging to the PARSEC benchmark suite [50, 42], the CPU utilization of the Apache HyperText Transfer Protocol (HTTP) Server and the IBM Lotus Domino Server [96].

We employ the linear discrete-time thermal model (simply thermal model from now on) described in Equation (11). $T(k)$ and $T(k+1)$ are the temperatures at the k-th and $(k+1)$-th sample instants, respectively; $I(k)$ is the idle time between the k-th and $(k+1)$-th sample instants; while a and b are parameters defining the temperature behavior.

$$T(k+1) = a \cdot T(k) + b \cdot I(k) \tag{11}$$

According to the thermal model, we can approximate the future temperature by accounting for its current value and the idle time between the current and future time instants.

Figure 35 shows a thermal simulation leveraging the compact thermal model proposed by Skadron et al. [83]. We simulate a worst-case application capable of pushing temperature of an "abstract" single-core processor up to 80 °C given an idle temperature of 30 °C (see the red line labeled "w/o ICI" in

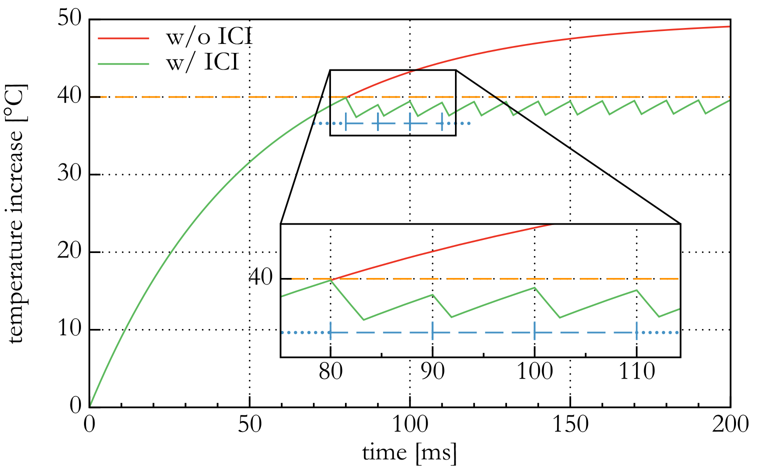


Figure 35.: Thermal simulation leveraging the compact thermal model giving visual evidence on the applicability of the linear discrete-time thermal model.

Figure 35). One should note that the temperature behavior has been artificially accelerated to show temperature oscillations.

The thermal simulation comprises idle cycle injection for DTM to alternate high with low power consumption phases. The control period for idle cycle injection is set to 10 ms (see the ticks on the blue dashed line in Figure 35) while the idle time can reach at most 80 % of this value (*i.e.,* 8 ms). The temperature threshold is set to 70 °C while the trigger threshold is set to 1 °C less. On the first thermal emergency at 80 ms, ThermOS injects ≈ 30 % of idle time and, from that point on, it injects ≈ 20 % of idle time per control period to keep temperature around the threshold (see the green line in Figure 35).

The simulation gives visual evidence that the temperature behavior around the threshold when employing idle cycle injection for DTM is quasi-linear both when idle cycle injection does and does not inject idle time and provides a first hint on the applicability of our thermal modeling approach.

4.3.2 *Thermal Model Training*

The estimation of parameters can be performed either online or offline and combinations of the two may apply. Online estimation is beneficial for time-variant workloads alternating CPU with memory and I/O-intensive phases and for workloads with CPU-intensive phases in which the Instructions Per Cycle (IPC) rate has a high variance, since it allows a thermal model to better track the temperature behavior. However, online estimation introduces overhead at run time since usually the better the estimation algorithm the higher its execution time. Offline estimation is suited for steady time-invariant workloads characterized by a single phase that is either CPU-, memory-, or I/O-intensive. Since the estimation of parameters occurs offline, the run time overhead is completely absent. Offline estimation requires an accurate training phase to guarantee that a thermal model fits the temperature behavior.

We decided to use offline estimation for the following three reasons:

1. we focus on multi-programmed CPU-intensive workloads that have a high probability of increasing temperature;
2. we use a reasonably high control frequency in the realm of operating systems and hence we must keep the temporal overhead under control;
3. we execute in kernel-mode and hence we are discouraged from using floating point computation; this makes almost prohibitive the implementation of most estimation algorithms at run time.

4.3.3 *Estimation and Empirical Validation*

We setup a modified version of ThermOS on our evaluation platform; this version of ThermOS randomly selects a value for the idle time in the interval $[0\,\%, 80\,\%]$. We run a worst-case workload consisting of four instances of *cpuburn* [97] to make the linear discrete-time thermal model conservative with respect to real-world workloads. We collected about 1.5 millions of triples—equivalent to $\approx 1\,\mathrm{h}$ of execution—consisting of the current temperature value, the previous temperature value, and the idle time value to catch most of the temperature behavior.

We used the least squares algorithm to estimate the parameters and fit the linear discrete-time thermal model reported in Equation (11). The least squares algorithm "solves" the linear system $y = X \cdot w$, where y is the column vector made up of n values of the current temperature, X is the matrix of n tuples consisting of the previous temperature value and the idle time value, and w is the column vector containing the m parameters (*i.e.,* a and b).

We partitioned the collection of triples in two: a training set of 70 % of the triples and a validation set of 30 % of the triples. We run the algorithm on the training set and verified the result against the validation set. The trained linear discrete-time thermal model yields a correct prediction for over 95 % of the triples of the validation set. Although we expect lower accuracy with real-world workloads due to the conservative nature of our thermal modeling approach, it is still accurate enough. We iterated the estimation many times with different training and validation sets to gain information about the robustness of parameters estimation. We eventually selected the best couple of parameters for our evaluation platform where a and b are 1.0244 and -0.0484, respectively.

While the simulation of ThermOS shows that the linear discrete-time thermal model is meaningful, the empirical validation gives mathematical evidence and strengthen to our thermal modeling approach.

4.3.4 *Statistical Validation*

We further evaluated the quality of the estimated parameters values through the computation of their statistical significance. Since we used the least squares algorithm, it was possible to estimate the variance of the parameters by means of the statistic reported in Equation (12). w_i is the i-th parameter; X is the matrix of coefficient of the linear system "solved" by the least squares algorithm; S is the sum of squared residuals computed according to Equation (13); n and m are the number of triples used by the least squares algorithm and the number of parameters of the linear discrete-time thermal model, respectively.

$$\mathrm{Var}(w_i) = \sigma^2([X^T \cdot X]^{-1})_{ii} \approx \frac{S}{n-m} \cdot ([X^T \cdot X]^{-1})_{ii} \tag{12}$$

$$S = \sum_{i=1}^{n} (y_i - X_i \cdot w)^2 \tag{13}$$

The estimated variances of the a and b parameters values across different training/validation partitions are $6.7851 \cdot 10^{-8}$ and $1.4059 \cdot 10^{-7}$, respectively. They suggest our thermal modeling approach is robust.

4.4 THERMAL MANAGER

We structured ThermOS following a feedback design and implemented it inside the Linux kernel. The first step consists in observing temperature values. The second step takes decisions regarding the needed idle time. Finally, the third step incorporates the idle time into applications execution.

4.4.1 *Temperature Measurement*

Formal feedback control requires contextual information; more specifically, DTM leverages temperature "measurements". One can either rely on analytic thermal models or employ thermal sensors to provide such "measurements".

ThermOS provides DTM through a software solution that mitigates the drawbacks of hardware DTM. However, software DTM cannot provide strong guarantees of limiting temperature; thankfully, ThermOS can still rely on hardware DTM to face thermal emergencies. Because of this reason, the low resolution of temperature measurements obtained through on-chip Digital Thermal Sensorss (DTSs) available on commodity CMPs are sufficient. For example, our evaluation platform features an Intel Xeon Processor W3530 supporting per-core DTSs with a resolution of 1 °C.

We implemented the observation phase through high-priority kernel-mode threads. Each high-priority kernel-mode thread always executes on the same core and periodically probes machine-specific registers to retrieve the temperature measurement of the core it is running on.

4.4.2 *Idle Time Determination*

Formal feedback control is successful in managing systems explained through mathematical models. The physical laws ruling thermal phenomena provide a strong mathematical model enabling the use of formal feedback control for DTM, thus avoiding the difficulties associated with specially-designed controllers.

Formal feedback control provides many advantages thanks to its formalism. It is possible to design controllers with predictable behavior in terms of response time and to achieve desirable stability and robustness guarantees.

Control theory helps synthesizing controllers that achieve the desired output by exploiting the availability of mathematical models of the controlled processes. In particular, industry strongly relies on formal feedback control and harnesses well-known solutions that proved beneficial even when dealing with approximate mathematical models: P, PI, and PID (proportional-integral-derivative) controllers.

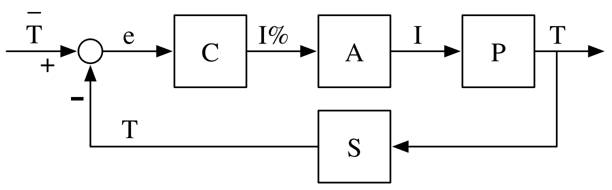


Figure 36.: Feedback design of ThermOS.

Formal Feedback Controller

Figure 36 shows a feedback design, where the sensor S measures the output T of the process or plant P; the controller C computes the error e subtracting the output T from the desired output $\bar{T}$ and then constraints process the input I through the actuator A. Following this paradigm, we realize per-core formal feedback controllers.

Let T_{emerg} be the temperature threshold such that exceeding such value causes a thermal emergency. We must start DTM before reaching T_{emerg}, so we set a temperature trigger threshold $\bar{T} < T_{emerg}$. We periodically sample the temperature measurement $T_i(k)$ of core i following the methodology reported in Section 4.4.1. We compute the error $e_i(k) = \bar{T} - T_i(k)$. If the error $e_i(k)$ is negative the i-th core is overheating. Otherwise, if the error $e_i(k)$ is positive, the i-th core is working properly.

We devised a PI controller that responds to errors by means of two terms: (1) a proportional term and (2) an integral term. The proportional term changes its effect according to the current value of the error and in a way that decreases the future values of the error. The integral term changes its effect incrementally accounting for the past values of the error. We neglected the derivative term; while this results into a little loss of control, at the same time it leads to notably less noise.

The synthesis of the PI controller depends on whether the mathematical model of the process is continuous-time or discrete-time. Since we periodically obtain per-core temperature measurements with a specified sample period, the mathematical model reported in Equation (11) is discrete-time. Thus, we perform the derivation in the $\mathcal{Z}$-transform domain.

Equation (14) represents the PI controller for core i, where $I_i(k)$ is the idle time required to constrain temperature, expressed as a percentage of the control period.

$$I_i(k) = I_i(k-1) + \frac{1-p}{b} \cdot e_i(k) - a \cdot \frac{1-p}{b} \cdot e_i(k-1) \tag{14}$$

Controller Synthesis and Stability Proof

We devised a PI controller for the linear discrete-time thermal model reported in Equation (11) with the goal of keeping temperature $T(k)$ as close as possible to the trigger threshold $\bar{T}$.

We determined the transfer function $\mathcal{P}(z)$ applying the $\mathcal{Z}$-transform to the linear discrete-time thermal model reported in Equation (11). Equation (15) shows the result, where $\mathcal{T}(z)$ and $\mathcal{I}(z)$ are the $\mathcal{Z}$-transforms of temperature and idle time, respectively.

$$\begin{gathered} z \cdot \mathcal{T}(z) = a \cdot \mathcal{T}(z) + b \cdot \mathcal{I}(z) \\ \mathcal{P}(z) = \frac{\mathcal{T}(z)}{\mathcal{I}(z)} = \frac{b}{z-a} \end{gathered} \tag{15}$$

We synthesized the PI controller by constraining the transfer function of the feedback as explained by Levine [98]. Equation (16) holds the result; $\mathcal{G}(z)$ and $\mathcal{C}(z)$ are the transfer functions of the feedback and of the PI controller, respectively.

$$\mathcal{G}(z) = \frac{\mathcal{C}(z) \cdot \mathcal{P}(z)}{1 + \mathcal{C}(z) \cdot \mathcal{P}(z)} = \frac{1-p}{z-p} \tag{16}$$

We employed a first order transfer function with a pole in p, a configurable parameter whose value changes the responsiveness of the PI controller. If p is chosen in the interval $(-1, 1)$ the feedback is asymptotically stable. Moreover, if p is chosen in the interval $(0, 1)$ the feedback guarantees the absence of oscillations. Given asymptotic stability and the absence of oscillations for granted, large values of p in the interval $(0, 1)$ translate into a slower but smoother response, while small values of p translate into a faster but rougher response. ThermOS provides a compile time setting to change the value of p in the interval $(0, 1)$, therefore we conclude the feedback is asymptotically stable.

Starting from Equation (16), we determined the transfer function $\mathcal{C}(z)$ of the controller; Equation (17) holds the result.

$$\mathcal{C}(z) = \frac{(1-p)\cdot(z-a)}{b\cdot(z-1)} \tag{17}$$

We imposed $\mathcal{C}(z) = \mathcal{I}(z)/\mathcal{E}(z)$; this leads to the transfer function reported in Equation (18).

$$\begin{aligned} \frac{\mathcal{I}(z)}{\mathcal{E}(z)} &= \frac{(1-p)\cdot(z-a)}{b\cdot(z-1)} \\ z\cdot\mathcal{I}(z) - \mathcal{I}(z) &= z\cdot\frac{1-p}{b}\cdot\mathcal{E}(z) - a\cdot\frac{1-p}{b}\cdot\mathcal{E}(z) \end{aligned} \tag{18}$$

The $\mathcal{Z}$-antitransform and a time shift applied to Equation (18) yield Equation (19), the generic form of Equation (14).

$$I(k) = I(k-1) + \frac{1-p}{b}\cdot e(k) - a\cdot\frac{1-p}{b}\cdot e(k-1) \tag{19}$$

Controller Robustness Analysis

In general, a controller depends on the process it is responsible for. In our case, the PI controller depends on the linear discrete-time thermal model depicted in Equation (11), whose parameters were estimated and validated in Section 4.3.2.

The statistical significance of the a and b parameters values makes us confident. However, ThermOS must deal with many events that can suspend idle cycle injection; this negatively impacts the effectiveness of idle cycle injection, which is represented by the b parameter. For this reason, we ask ourselves: what if our thermal modeling approach is not faithful and, in particular, what if the b parameter of the linear discrete-time thermal model is poorly estimated? In other terms, what if the weight of $I(k)$ is not b but $b + \delta$? We answer this question by means of a robustness analysis.

We assume the real process is described by the transfer function $\mathcal{P}(z)$ reported in Equation (20).

$$\mathcal{P}(z) = \frac{b + \delta}{z - a} \tag{20}$$

We substitute Equation (20) in Equation (16). The pole of the transfer function of the feedback changes from $z = p$ to Equation (21).

$$z = \frac{p \cdot (b + \delta) - \delta}{b} \tag{21}$$

If the pole lays in the interval $(-1, 1)$ the feedback remains asymptotically stable and loses at most the guarantee on the absence of oscillations. We solve the system of inequalities that leads to Equation (22).

$$\delta > \frac{b \cdot (1 + p)}{1 - p} \quad \wedge \quad \delta < -b \tag{22}$$

In practice, δ can vary in the interval $(-0.0899, 0.0484)$ when the b and p parameters values are -0.0484 and 0.3, respectively. The interval is large when compared to the estimated value of the b parameter. Hence, we declare that ThermOS is robust with respect to estimation errors on the effectiveness of idle cycle injection.

4.4.3 *Idle Cycles Injection*

The scheduling infrastructure of the Linux kernel enables different algorithms to schedule different types of threads (*i.e.,* either a process or a thread). This materializes in different scheduling classes with different priorities. The scheduling skeleton iterates over scheduling classes from the highest to the lowest priority to pick the next runnable thread.

The scheduling infrastructure of the Linux kernel provides five scheduling classes: (1) `SCHED_FIFO`, (2) `SCHED_RR`, (3) `SCHED_OTHER`, (4) `SCHED_BATCH`, and (5) `SCHED_IDLE`. The first two scheduling classes provide "real-time" policies while the remaining provide "normal" policies.

We implemented idle cycle injection for user-mode threads scheduled through normal policies, which are under the control of the Completely Fair Scheduler (CFS). This is consistent with previous implementations of idle cycle injection [17]. The rationale behind this choice is avoiding the preemption of real-time threads, which are rarely present in most GNU/Linux systems, and kernel-mode threads, which usually run with low IPC, causing low power consumption and temperature.

When the scheduling skeleton calls CFS, ThermOS enters the actuation phase and may or may not perform idle cycle injection depending on the outcome of the decision phase. We implemented idle cycle injection within the Linux kernel exploiting the availability of an idle thread for each core. CFS eventually picks the idle thread instead of the next runnable thread and runs it for as long as the thermal controller (*i.e.,* the PI controller) dictates.

Changing the Idle Task

The Linux kernel executes the idle thread whenever there are no runnable threads; the idle thread yields as soon as a thread becomes runnable. ThermOS also executes the idle thread whenever the thermal controller demands the injection of idle time.

Without loss of generality, we analyze the behavior of the Linux kernel when executing on top of Intel x86 and x86-64 processors. The Linux kernel runs the idle thread in kernel-mode to allow the execution of protected instructions. The idle thread issues the `MONITOR` instruction to arm the address monitoring hardware with the address of the `flags` variable stored in its `task_struct`. It then issues the `MWAIT` instruction to request the processor entering the C1E state.

The Linux kernel eventually writes the `flags` variable of the idle thread to demand a reschedule. The address monitoring hardware catches the write operation forcing the processor to exit the C1E state and enter the C0 state. Finally, the `MWAIT` instruction returns and the idle thread yields.

We modified the idle thread to issue the `MONITOR` instruction to arm the address monitoring hardware with different variables depending on the outcome of the thermal controller. Whenever the thermal controller demands the injection of idle time, the idle thread selects the `thermal_flags` variable, which is once again stored in its `task_struct`. Otherwise, the idle thread selects the `flags` variable and its behavior is unmodified. The idle thread then issues the `MWAIT` instruction to request the processor entering the C1E state.

The Linux kernel eventually writes either the `thermal_flags` or the `flags` variable of the idle thread to indicate the idle time is exhausted or a runnable thread is available possibly triggering a C-state transition. In addition, the Linux kernel writes the `thermal_flags` instead of the `flags` variable of the idle thread whenever the idle thread is running for cooling purpose and either a real-time or a kernel-mode thread became runnable. This grants ThermOS with the capability of suspending idle cycle injection to face the execution of real-time or kernel-mode threads.

Exploiting the Dynamic Tick

The Linux kernel uses a periodic timer firing at a configurable frequency—100, 250, 333, 1000 Hz—for "house-keeping" operations. This timer is usually referred to as *scheduler tick*.

The scheduler tick forces the processor to exit low power states and hence increases the energy consumption even when the Linux kernel is executing the idle thread. Since energy consumption is a fundamental issue, the Linux kernel 2.6.21 introduced the *dynamic scheduler tick*. The scheduler tick is temporarily disabled to "idle" instead of "idle with ticks". The scheduler tick fires periodically whenever the Linux kernel executes runnable threads while it fires on-demand whenever the Linux kernel executes the idle thread.

When the Linux kernel sets the scheduler tick to fire on-demand it takes the difference between the current time and the next time a software interrupt request must execute. We modified this behavior by choosing the minimum between the next time a scheduled interrupt request (*e.g.,* `sleep(2)`) must execute and the idle time.

Scheduling the Idle Task and Alternatives

In the remainder of this section we analyze alternative approaches to scheduling the idle thread as a means for idle cycle injection and we highlight the choices that led to our design.

Scheduling the idle thread as a means for idle cycle injection may be sub-optimal from a performance standpoint: it requires *trapping* from user to kernel-mode and *context switching* the current thread with the idle thread. A first alternative approach targets the context switching issue. One could avoid the cost of context switching from the current thread to the idle thread by "replicating" the functionality of the idle thread and the *cpuidle* infrastructure: issue the `MONITOR` and `MWAIT` protected instructions that allow the processor to arm the address monitoring hardware and transition from the C0 to the C1E state. This approach violates the basics of software engineering by replicating a well-structured functionality.

A second alternative approach targets both the trapping and the context switching issues. One could avoid the cost of trapping from user to kernel-mode and hence the cost of context switching from the current thread to the idle thread by issuing "low-power" instructions in user-mode. The adoption of a just-in-time (JIT) compiler allows changing the code of an application at run time; it is theoretically possible harnessing this feature to "inject" a series of `NOP` instructions to cool down the processor. Unfortunately, the effectiveness of this approach is limited by design by the use of the `NOP` instruction, which does not allow the processor to transition from the C0 to the C1E state. We are aware of state-of-the-art compiler-directed techniques to decrease the peak temperature of a processor to improve long-term reliability [99]; however, these techniques target the mitigation of long-term effects like the negative bias temperature instability and the aging.

We quantified the overhead of trapping and context-switching: it is limited between 3 and 30 µs where the worst case accounts for thread migration. We concluded that the overhead is acceptable and does not compromise the efficiency of ThermOS.

4.5 evaluation

This section evaluates ThermOS and, in particular, it is focused on answering the following questions:

1. can ThermOS constrain temperature and affect applications within a multi-programmed workload depending on cores thermal profiles?

2. how efficient is ThermOS in tackling the trade-off between performance and temperature when compared to *Dimetrodon* and DVFS?

4.5.1 *Evaluation Platform and Configuration*

We evaluated ThermOS on a Dell Precision T3500 workstation with an Intel Xeon Processor W3530 and 12 GB of main memory in 3 memory modules. The processor features 4 cores operating at 2.80 GHz and sharing 8 MB of Last-Level Cache (LLC). Each memory module runs at 1066 MHz. The Enhanced Intel SpeedStep Technology allows the processor to work in ten different P-states from 1.60 to 2.80 GHz. We disabled the Intel Turbo Boost Technology to prevent the processor entering P-states with clock frequency higher than the nominal when a subset of the cores is executing. The Intel Turbo Boost Technology would bias the analysis in favor of ThermOS that creates unbalanced execution times since it exploits the different thermal profiles. We disabled the Intel Hyper-Threading Technology (HTT) to simplify our implementation. When HTT is enabled, each physical core is in fact a couple of virtual cores requiring idle cycle injection co-scheduling to enter the C1E state [90].

We configured Debian 7.0 to run the Linux kernel 3.4 enhanced with ThermOS and FreeBSD 7.2 enhanced with *Dimetrodon* [17]. We configured ThermOS with a temperature sampling period of 10 ms and a control period of 10 ms. The thermal controller was setup to limit the idle time to 80 % of the control period and the temperature trigger threshold is 3 °C lower than the temperature threshold. The a, b, and p parameters values are 1.0244, −0.0484, and 0.3, respectively. Section 4.3.2 supports the choice of these values.

We assessed the behavior of ThermOS through the PARSEC 2.1 benchmark suite [50, 42], which provides a set of representative workloads. We ran multi-programmed workloads of single-threaded applications pinned to cores. Section 4.5.4 comments on ThermOS's behavior with multi-threaded applications.

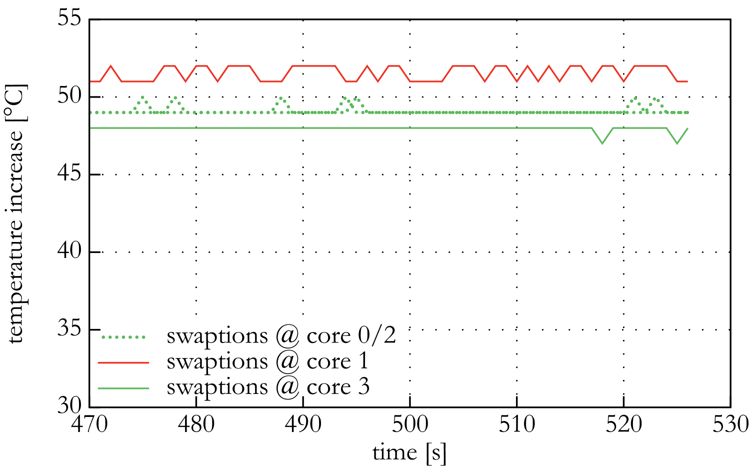


Figure 37.: Executions of a multi-programmed workload on a commodity CMP. Any form of DTM is disabled, hence temperature rises without constraints.

4.5.2 *Addressing Multi-Programmed Workloads*

We first show how ThermOS is capable of constraining temperature and affecting applications within a multi-programmed workload depending on cores thermal profiles.

We thoroughly explain the behavior of ThermOS when running a homogeneous multi-programmed workload consisting of four instances of *swaptions*, each one running with a single thread of execution on its own core since we believe it helps making our point. However, similar considerations hold for the various multi-programmed workloads we employ in the remainder of this chapter.

The multi-programmed workload leads to a steady temperature of about 80 °C with an idle temperature of 30 °C. Figure 37 shows the last minute of execution without any form of DTM and highlights different thermal profiles for the four cores; core 1 operates at a higher temperature than the other cores and overcomes 80 °C while core 3 operates at a lower temperature. Figure 38

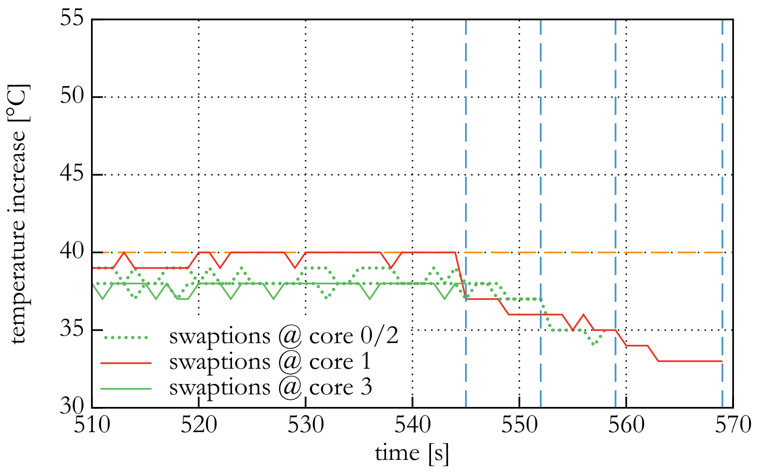


Figure 38.: Executions of a multi-programmed workload on a commodity CMP. ThermOS is enabled, hence temperature rises but remains constrained below the threshold.

displays the last minute of execution with ThermOS configured to constraint temperature below 70 °C. ThermOS enforces the threshold. Table 9 breaks down the execution time of the multi-programmed workload per core per C-state when executing on ThermOS. Each instance of *swaptions* always requires the execution of the same amount of instructions to run to completion and in fact they complete at the same time when any form of DTM is disabled (see Figure 37). All cores spend approximately the same percentage of the execution time in C0 since it is the only C-state in which cores actually execute instructions of the instances of *swaptions*.

When executing on ThermOS, the real execution time of an instance of *swaptions* accounts for the time spent in C0 and C1E since ThermOS exploits only the latter to lower temperature instead of using C3 and C6 that introduce higher latency to enter and exit the C-state (*i.e.,* 20 and 200 µs, respectively, instead of 3 µs) and penalties (*e.g.,* private caches and register file flushes). Cores spend different percentages of the execution time in C1E and C3 since ThermOS injects idle time depending on cores thermal profiles and hence the instances of *swaptions* complete at different instants. For example, core 1 oper-

Table 9.: Break down of the execution time of a multi-programmed workload per core per C-state with ThermOS

Core	C0	C1E	C3	C6
0	91 %	7 %	2 %	0 %
1	91 %	9 %	0 %	0 %
2	91 %	6 %	3 %	0 %
3	91 %	5 %	4 %	0 %

ates at a higher temperature than the other cores and the instance of *swaptions* it runs is the last to complete. Thus, core 1 spends the highest percentage of the execution time in C1E when compared to the other cores and none in C3. Conversely, core 3 operates at a lower temperature and the instance of *swaptions* it runs is the first to complete. Thus, core 3 spends the lowest and the highest percentages of the execution time in C1E and C3, respectively, when compared to the other cores.

Figure 38 displays the behavior of ThermOS when executing a CPU-intensive workload. However, ThermOS is independent from the kind of workload. Figure 34 highlights the same capabilities even when ThermOS executes a workload consisting of a CPU and an I/O-intensive applications.

4.5.3 *Addressing the Performance/Temperature Trade-Off*

We also show how ThermOS is efficient in tackling the trade-off between performance and temperature.

We configured ThermOS to achieve temperature decreases of: 10, 20, 25, 30, and 35 % with respect to the idle temperature. We configured *Dimetrodon* varying the idle time between 10 and 100 ms and the idle probability between 0 and 40 % for a total of 150 configurations. We statically set the following P-states: 2.79, 2.66, 2.53, 2.39, 2.26, and 2.13 GHz through DVFS. Each P-state is used for the whole execution of a multi-programmed workload.

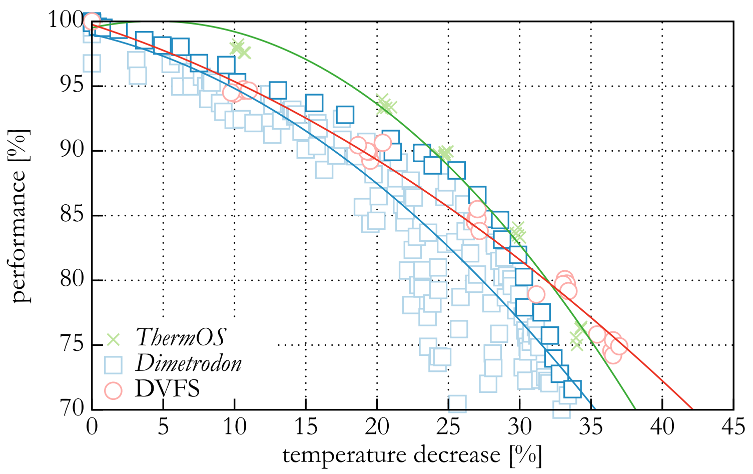


Figure 39.: Efficiency in tackling the trade-off between performance and temperature for ThermOS, *Dimetrodon*, and DVFS.

We ran various multi-programmed workloads consisting of four applications among: *blackscholes*, *bodytrack*, *canneal*, *dedup*, *ferret*, *fluidanimate*, *raytrace*, *streamcluster*, *swaptions*, and *x264* and balanced their run times by re-execution.

Figure 39 displays the efficiency curves of ThermOS, *Dimetrodon*, and DVFS. Performance (*i.e.*, the ratio between the execution time without and with the intervention of DTM techniques) decreases linearly from 100 % to 75 % when setting the P-states through DVFS; however, temperature decreases are less predictable.

Dimetrodon employs a probabilistic, feedforward controller to drive idle cycle injection; the probabilistic nature of the controller and the absence of feedback make the behavior mostly unpredictable. Figure 39 highlights the Pareto-optimal executions of *Dimetrodon*, which are slightly worse than those of ThermOS; however, the interpolation of all the executions is worse than that of DVFS. Conversely to *Dimetrodon*, ThermOS employs a formal feedback controller backed by a robust thermal model to drive idle cycle injection. The conjunction of these elements make the behavior of ThermOS highly predictable. Figure 38 displays a performance decrease of $\approx 8\,\%$ with respect to Figure 37; this is expected considering Figure 39 and a temperature decrease of $\approx 20\,\%$.

When cores operate at decreased clock frequency the relative latency of the memory hierarchy tends to decrease alongside with the bandwidth [100]. While the latter effect is negative and compromises the performance of memory-intensive applications, the former is positive for CPU-intensive applications since it lessens the effects of memory stalls. We believe P-states close to the highest do not allow DVFS to balance the number of instructions issued, which is known to have a higher correlation with temperature than the number of instructions retired [101], with an adequate decrease of power consumption.

It is well accepted that the dynamic power consumption of an integrated circuit made up of an ensemble of transistors scales as reported in Equation (23); where a is some proportionality constant, C is the capacitance of a single transistor, V_{dd} is the supply voltage, f is the clock frequency, and n_t is the number of transistors that switches concurrently on average.

$$P_d \propto a \cdot C \cdot V_{dd}^2 \cdot f \cdot n_t \tag{23}$$

Until the beginning of 2000s, the supply voltage V_{dd} has decreased constantly with the ever-decreasing feature size; however, the shift from the micrometer to the nanometer realm prevents this from happening with the same pace as before. The supply voltage V_{dd} is bound to be twice as much as the threshold voltage V_{th}, which is not scaling down [102]. DVFS is doomed to progressively lose its effectiveness since the clock frequency f will be the only difference among the available P-states and its weight is not comparable to that of the squared supply voltage V_{dd}^2. ThermOS already achieves better efficiency than DVFS when tackling the trade-off between performance and temperature for decrease of the latter up to $35\,\%$ and this margin is likely to increase in the foreseeable future. In fact, future efficiency curves for DVFS will most likely fall in the area below the current one (see Figure 39).

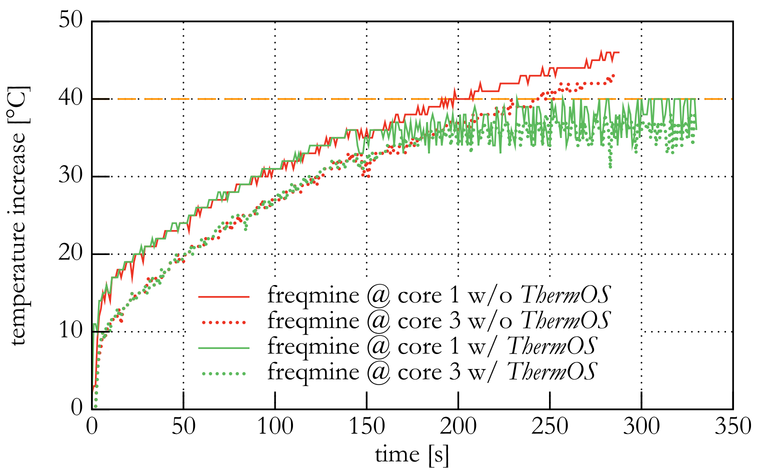


Figure 40.: Execution of a multi-threaded application on a commodity CMP. ThermOS constraints temperature below the threshold but introduces artificial critical paths making performance unpredictable.

4.5.4 *Limitations*

In this work we focus on the trade-off between performance and temperature for multi-programmed workloads consisting of single-threaded applications. Let us consider a different setting in which a CMP runs a multi-threaded application. As already pointed out, cores on a CMP heat up following different thermal profiles, thus requiring a more or a less aggressive idle cycle injection. Whenever this happens, one or more threads may be slowed down more than the others, thus putting in place an artificial critical path impairing synchronization and stretching the run time unpredictably. Figure 40 displays this issue by means of two consecutive runs of *freqmine* executing with four threads. In this "unsupported" setting, ThermOS achieves a temperature decrease of $\approx 11\,\%$ at the cost of a performance decrease of $\approx 15\,\%$.

Being aware of this issue we plan on improving ThermOS so as to cope with synchronizations. A first naïve approach for finely-synchronized (*e.g.*, barrierbased) multi-threaded applications requires idle cycle injection to work with the same timing and intensity among the cores running an application, doing so by choosing the idle time required by the core with the worst thermal profile. This is clearly sub-optimal from a performance standpoint. A more elaborated solution requires an augmented thermal model accounting for thermal interactions among cores since synchronizing idle cycle injection will greatly enhance its efficiency [101, 92]. An approach for coarsely-synchronized (*e.g.*, lock-based) multi-threaded applications exploits previous work on scheduling for symmetric multi-processors and avoids throttling a thread inside a critical section.

4.6 related work

Researchers proposed a variety of techniques to deal with temperature issues. We classify these techniques in three categories: architectural, micro-architectural, and software.

4.6.1 *Architectural Approaches*

Clock and power gating limit the distribution of the clock signal and the supply voltage, respectively. They decreases the dynamic and static power consumption, respectively, since the former prevents transistors from switching while the latter cut them off from the supply voltage. C-states exploit clock and power gating to decrease energy consumption.

Near/sub-threshold voltage (NTV/STV) designs [103] dramatically increase energy efficiency at the cost of severe drops of clock frequencies and single-threaded performance. With the shift from single to multicore processors we assisted to the proliferation of multi-threaded applications. The adoption of NTV/STV designs moves the limits even further requiring embarrassingly multi-threaded applications, which are far from common. Hence, researchers disagree about the applicability and success of NTV/STV designs [102].

A recent architectural approach employs *c-cores* [104] to decrease energy consumption and power density by means of pre-synthesized application-specific co-processors. *c-cores* are promising but require a substantial paradigm shift at the computer architecture level since the most likely implementation require the adoption of reconfigurable fabrics.

4.6.2 *Micro-Architectural Approaches*

Micro-architectural approaches like instruction window sizing, issue width sizing, and instruction fetch toggling aim at limiting energy consumption by decreasing the number of instructions issued per clock cycle. Performance penalties due to the adoption of micro-architectural approaches can be amortized by orthogonal techniques like activity migration [105], which however requires additional transistors.

Brooks and Martonosi [93] propose a set of heuristics to drive instruction fetch toggling. Skadron et al. [95] show the applicability of the compact thermal model and formal feedback control to drive instruction fetch toggling achieving predictable behavior and the desirable properties control theory can guarantee. Jayaseelan and Mitra [106] harnesses instruction window and issue width sizing alongside with instruction fetch toggling and a neural network predictor to implement DTM.

Collectively, micro-architectural approaches can guarantee safety; however, being implemented at the lowest level of the hardware/software execution stack, they lack visibility and may impair the performance of critical pieces of software such as real-time and kernel-mode tasks, and interrupt request routines. In addition, most of these approaches are not available in commodity designs that need alternative software approaches.

4.6.3 *Software Approaches*

Common software approaches regard thermal-aware scheduling and DTM. Thermal-aware scheduling for large-scale computing systems involves migrating tasks from hot to cold islands [107], while thermal-aware scheduling for servers is concerned with tasks placing [101] and ordering [94].

Powell et al. [101] propose *Heat-and-Run*, a technique to perform assignment and migration of tasks to balance temperature across a CMP. *Heat-and-Run* is a thermal-aware scheduler that exploits Simultaneous Multi-Threading (SMT) to co-schedule "complementary" tasks (*e.g.,* one ALU-intensive and one FPU-intensive) on the same core and the availability of many cores to alternate heating and cooling phases.

Zhou et al. [94] propose *ThreshHot* and observes that tasks ordering actually matters. *ThreshHot* is a thermal-aware scheduler that orders tasks from "hot" (mostly CPU-intensive) to "cold" (mostly I/O-intensive) and schedules them from the hottest to the coldest. This schedule is guaranteed to minimize temperature at the end of an epoch. The goal of thermal-aware scheduler is minimizing temperature without degrading vital measures such as throughput and latency.

DTM, and hence ThermOS, is orthogonal to thermal-aware scheduling since the former tackles those settings in which the latter cannot prevent temperature from exceeding the threshold. Kumar et al. [108] propose *HybDTM*, which still exploits the "hot" and "cold" tasks classification but without following a "hot-to-cold" schedule. Whenever temperature exceeds the threshold, *HybDTM* throttles "hot" tasks first by lowering their priority, thus allowing "cold" tasks to use more processor time. *HybDTM* is meant for single-core processors and many of its considerations do not apply in the CMP realm.

kidled [91] is Google's idle cycle injection implementation. It allows administrators to set a core-wide idle time over a time period. If the end of an interval draws near and the core has not been naturally idle for the requisite time, *kidled* injects idle time. *PowerClamp* [92] is Intel's idle cycle injection implementation. Bailis et al. [17] propose *Dimetrodon*, a framework implemented inside the FreeBSD kernel that relies on probabilistic feedforward control and idle cycle injection as a means for decreasing temperature. Conversely to ThermOS, *kidled*, *PowerClamp*, and *Dimetrodon* are eager when injecting idle time. In addition, ThermOS leverages a thermal model and formal feedback control to drive idle cycle injection.

4.7 SUMMARY AND FUTURE WORK

ThermOS proved effective in managing temperature during the execution of multi-programmed workloads and achieved better efficiency than both commodity and cutting edge DTM techniques. The evaluation shows that ThermOS can selectively affect applications within batch-style, multi-programmed workloads running on a commodity CMP. ThermOS also displays higher flexibility and better efficiency than DVFS for temperature reduction of up to 30 %. Moreover, the use of formal feedback control provides ThermOS with better predictability than *Dimetrodon*.

As future work we intend to both address the limitations highlighted in this paper (*e.g.*, exploit technologies like SMT and Hyper-Threading Technology (HTT), execute multi-threaded applications, etc.) and integrate DTM with performance management [8, 12, 9] to guarantee service-level objectives.

5 ADAPTIVE PERFORMANCE AND THERMAL MANAGEMENT

In modern computing facilities, higher and higher operating temperatures are due to the employment of power-hungry devices, hence the need for cost-effective heat dissipation solutions to guarantee proper operating temperatures. Within this context, Dynamic Thermal Management (DTM) can be highly beneficial in proactively control heat dissipation, avoiding overheating. The large-scale adoption of DTM may eventually allow the use of more cost-effective heat dissipation system, with great power consumption advantages for large datacenters.

Preventive thermal management is a technique to achieve long-term thermal control via performance degradation. However, this may result in impaired Service-Level Objective (SLO) and Service-Level Agreement (SLA) breaking. We address this problem by proposing a self-adaptive framework combining performance and thermal management targeting Chip-Multiprocessor (CMP). The proposed methodology harnesses control-theoretical controllers for driving idle cycle injection and threads priority adjustment, in order to provide control over the processor temperature, while taking applications' Quality of Service (QoS) (in terms of performance) into account. We implemented our framework in the FreeBSD operating system and evaluated it on real hardware, also comparing it with a previous framework for preventive DTM.

5.1 INTRODUCTION

Recently, Very-Large-Scale Integration (VLSI) design—in particular for multi-core processors—has been heavily influenced by the approach of the power wall. The shift from single to multi-core processors was a tentative response to power constraints, passing from one very complex processor to a set of simpler on-chip cores. However, even multi-core processors are hitting new barriers, originating the phenomenon of dark silicon [54]. Packing more and more transistors on a single die of the same size, with energy efficiency not scaling any more, results in an ever increasing power density, which requires more efficient cooling systems to keep proper operational thermal conditions. Maintaining processors cool is crucial for reliability and efficiency: higher average operating temperatures lead to a drastic reduction of the Mean Time To Failure (MTTF) [85] and highly increased leakage power [83]. Moreover, processors are acknowledged to be one of the most power-hungry and heat-producing components in a computing system and, for plenty of server workloads, power consumption is reported to increase almost linearly with processor utilization [109]. These considerations are of utter importance for data centers; up to 80 % of the Total Cost of Ownership (TCO)—including both building and maintenance costs—is due to the cooling infrastructure and the overall power consumption [110].

Recently, processor-level DTM techniques received quite a lot of attention and the trend is to move from runtime guards for emergency situations to always-on proactive control systems. Preventive DTM is a technique leveraging modern processors' features such as idling power states (C-states) in order to actively degrade performance for achieving thermal control [17]; however, doing so may cause problems when SLAs exist on the provided SLO.

We tackle this problem with a Dynamic Performance and Thermal Management (DPTM) framework based on self-adaptive computing [15], realizing preventive DTM while accounting for desired applications performance in terms of SLO specified by user-signed SLAs. The DPTM framework is an extension of a commodity operating system and it is able to inject idle cycles for cooling down cores and to vary threads' priority for affecting applications' performance. The main contributions of this work are:

- harnessing idle cycle injection to drive the processor temperature towards a desired set point;
- accounting for desired SLO and controlling threads' priorities for respecting SLAs on performance;
- coupling the thermal and the performance-aware mechanisms for both preventive DTM and SLAs enforcement.

We implemented DPTM extending FreeBSD 7.2 and the evaluation leverages applications from the PARSEC 2.1 benchmark suite [42].

5.2 RELATED WORK

DTM has been an active research topic in the recent years; some relevant works found in literature are surveyed in this section.

Rohou and Smith [111] presented an evaluation of the idle cycle injection. Since then, many researchers have exploited their early findings. Kumar and Thiele [112] proposed a DTM technique for real-time systems with the objective of minimizing the system peak temperature while meeting deadlines. The authors implemented a scheduler based on leaky-bucket shapers able to dynamically inject idle cycles and delay execution without violating deadlines. Experimental results show that the peak temperatures observed are 8.8 K lower with respect to those observed using a standard real-time scheduler. This technique is simple, effective, and able to manage any task arrival pattern; however, it is tailored to processors lacking Dynamic Voltage and Frequency Scaling (DVFS).

Gupta and Mahapatra [113] developed a DTM technique for real-time systems able to reduce violations of temperature constraints through DVFS and still meeting task deadlines. They implemented an on-line technique relying on a feedback controller that sets voltage and frequency so as to reduce thermal dissipation and meet deadlines. This can be done considering as input an upper-bound system temperature, the past system temperatures, and the jobs slacks. Simulation results show that constraint violations are reduced by 18.9 % on average.

Recently, Bailis et al. [17] proposed *Dimetrodon*, a framework for desktop and server systems realizing preventive DTM through idle cycle injection [111]. They argue that DVFS is an invasive technique and should be used only when relevant temperature variations are needed. Idle-cycle injection, instead, guarantees fine-grained control over the thermal dissipation of each running thread. *Dimetrodon* was implemented within the FreeBSD 7.2 kernel and evaluated on Central Processing Unit (CPU)-intensive workloads. The gathered data demonstrate that *Dimetrodon* obtains a better temperature-performance trade-off with respect to DVFS for temperature reductions of up to 30 % of the idle temperature. A drawback of *Dimetrodon* lies in the logic employed to inject idle cycles, which are inserted randomly without considering possible SLAs on the delivered SLO. Moreover, the system does not allow to define a set point for the processor temperature, but just the idle cycle injection probability.

The DPTM framework we propose builds upon the methodology harnessed by *Dimetrodon* and addresses its drawbacks, allowing to specify a temperature set point and coupling thermal-awareness with performance-awareness in order to avoid breaking SLAs.

5.3 METHODOLOGY

Typical scheduling infrastructures in commodity operating systems adhere to the race-to-idle approach: applications are run to completion in order to idle the system as soon as possible, thus increasing the throughput of applications. This approach is generally energy-efficient overall [86], but it leads to higher peak power draw and thus to higher maximum chip temperatures. To contrast

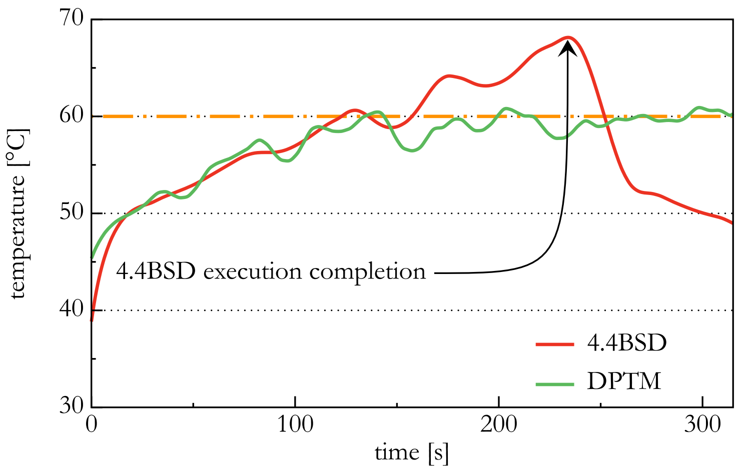


Figure 41.: Peak CMP temperature with a race-to-idle scheduler (4.4BSD) and the DPTM framework when executing the same CPU-bound workload. The temperature set point is 60 °C and the average peak CMP temperature with DPTM is 57.2 °C.

this—sometimes undesirable—property, whenever applications can afford a decrease in their throughput the scheduling infrastructure may exploit policies to reduce the peak power, thus reducing the maximum processor temperature and consequently requiring less power to operate the cooling infrastructure, the costs of which greatly impacts the management expenses, especially in the context of datacenters and server farms [114, 110]. Our proposed approach exploits this idea and Figure 41 shows an example execution highlighting the difference between a common race-to-idle approach and that proposed in this paper. Temperature reduction can be achieved by injecting idle cycles during the execution of a workload, letting the processor briefly go to a low power state, thus reducing the instantaneous absorbed power and, subsequently, the chip temperature. Randomly injecting idle cycles (as done in state-of-the-art approaches [17]), however, cannot be affordable when SLOs exist on the performance of some applications. Our methodology applies to this scenario, where CPU-bound applications are assigned high-level SLO goals in terms of desired throughput, similarly to what was proposed in different contexts [8]. Exploiting this knowledge, we can choose the victim tasks of the idle cycle injection process among applications not bound to an SLA, while not penalizing those with specified performance goals. Moreover, our framework also drives the scheduling priorities of SLO-bound applications in order to achieve finer performance control.

5.3.1 *Control-Theoretical Models*

To achieve control over peak and average processor temperature and CPU-bound applications performance, we extend the scheduling infrastructure of a commodity operating system with a thermal-aware and a performance-aware policy. The former monitors the current processor temperature and drives idle cycle injection to trigger the low-power mode in the cores, reducing the instantaneous absorbed power and, subsequently, the processor temperature. The latter observes user-defined throughput goals (*i.e.,* SLAs on the SLO of specific applications) and the current throughput, driving the scheduling priority of the threads of SLO-bound applications in order to increase or decrease the processor time they are ensured, thus affecting their performance.

When designing the models, we considered a trade-off between accuracy and complexity. Systems usually show non-linear behavior, but accounting for non-linearity in the model often reduces the provable properties and can impair the possibility of building a controller. We approximate non-linearity with linear models by devising a strategy to estimate parameters and to reduce the imprecision due to non-linearity.

Bailis et al. [17] already discussed the basics of idle cycle injection. Conversely to that work, which is based on a thermal-unaware probabilistic policy, we adopt a control-theoretical policy with a feedback loop, able to automatically drive the average processor temperature towards a specified set point, with the aim of cutting down the thermal peak typical of a race-to-idle approach. Let $T_i(k)$ be the temperature of the i-th core measured at time k; the goal of the controller is to stabilize $T_i(k)$ close to the set point $\tilde{T}$. We assume the model shown in Equation (24) represents the processor's thermal characteristics.

$$T_i(k+1) = T_i(k) + \mu_i \times \text{idle}_i(k) \tag{24}$$

The value $\text{idle}_i(k) \in [0\,\%, 100\,\%]$ represents the fraction of idle time injected in the i-th core during the time interval between the k-th and $k+1$-th sampling instants, while μ_i is an unknown parameter. The control-theoretical system, designed as an adaptive deadbeat controller [96], computes $\text{idle}_i(k)$ per core at each sampling instant. A deadbeat controller is synthesized so as the closed-loop transfer function equals a pure delay, *i.e.,* after one control step the set point $\tilde{T}$ should be transferred to the output temperature. It is possible to analytically demonstrate that, if μ_i is known, the set point will be attained [96] and the temperature will be kept at the reference level. Intuitively, idle cycles will be injected when the temperature gets too high, while no action will be taken while temperature remains lower than the set point. Since μ_i cannot be given a priori, we estimate it based on the last temperature measurements through an Exponential Moving Average (EWA) adaptive filter.

The performance-aware policy features another control-theoretical mechanism devoted to driving threads priorities; the policy requires user-defined goals to define SLAs and instrumented applications to provide throughput measures, similarly to what we proposed with Metronome [8]. SLO-bound applications are assigned a throughput goal stated as an interval $[r_{min}, r_{max}]$, where r_{min} is intended as the minimum performance satisfying the SLA and r_{max} is a level over which no significant SLO improvements are achieved. Legacy (*i.e.,* non instrumented) applications are assumed not to be bound to any SLA. Let $r_i(k)$ be the throughput of the i-th application at time k; the goal of the controller is to keep the performance of each instrumented application close to its set point $\tilde{r} = \frac{r_{min}+r_{max}}{2}$. We assume the performance model in Equation (25).

$$r_i(k+1) = r_i(k) + \eta_{i,j} \times \Delta\text{priority}_{i,j}(k) \tag{25}$$

The value $\Delta\mathrm{priority}_{i,j}(k) \in [-50, 50]$ represents the priority of the j-th thread of the i-th application, while $\eta_{i,j}$ is an unknown parameter. At each sampling instant, the control-theoretical system computes $\Delta\mathrm{priority}_{i,j}(k)$ per thread per application. Also in this case, the closed-loop system is designed to be a pure delay. As for the thermal-aware policy, if η_i is known then the set point signal will be attained. The Δpriority will be decreased whenever performance is getting higher than needed, while higher priority will be given when the desired SLO is not being met. Also in this case, the parameter $\eta_{i,j}$ is periodically estimated with an EWA adaptive filter.

5.3.2 *Performance-Temperature Trade Off*

Within the DPTM framework, a synergy is created between the thermal and performance-aware policies by means of a heuristic, which couples them for stronger performance control and finer thermal control. Since the performance-aware policy is in place to avoid breaking SLAs, this policy has precedence over the thermal-aware one. When an application bound to an SLA is running below its r_{min}, the performance-aware policy marks the application's threads to *prevent idle*. Conversely, when a controlled application is over performing (*i.e.*, its throughput is above r_{max}), its threads are marked to *force idle*. The thermal-aware policy uses these two flags to never charge idle time to threads of applications not respecting their SLA, while charging idle time—if needed—to threads of applications running faster than needed. In practice, performance is traded for temperature reduction only if no SLAs get broken.

5.4 IMPLEMENTATION

We implemented the proposed DPTM framework as an extension of FreeBSD 7.2, modifying the 4.4BSD scheduler.[1] The thermal-aware policy consists of a set

1 FreeBSD 7.2 supports two different schedulers: 4.4BSD and ULE. We chose 4.4BSD as a base for fair comparison with Dimetrodon [17].

of high-priority kernel threads measuring the temperature of cores and a high-priority kernel thread implementing the control-theoretical system described in Section 5.3.1, Equation (24). Temperature measurements are gathered by reading the appropriate Model-Specific Register (MSR) for each core, with an accuracy of $1°C$. Temperature constraints are defined through the `sysctl` utility. Similarly, the performance-aware policy is made up of a set of timers computing the throughput of applications and a high-priority kernel thread implementing the control-theoretical system described in Section 5.3.1, Equation (25). The infrastructure for throughput measurements is a complete port of the Heart Rate Monitor (HRM), proposed by Sironi et al. [8], to the FreeBSD 7.2 kernel.

The 4.4BSD scheduler is based on a multilevel feedback queues infrastructure: all runnable threads are assigned a priority determining the run queue they are placed on and threads are migrated according to their changing priority. In selecting the new thread to run, the scheduling infrastructure scans the run queues from the highest to the lowest priority and it chooses the first thread found on the first non-empty run queue. Multiple threads on the same run queue are managed in a round robin fashion and are assigned a fixed timeslice of $100\,\text{ms}$ [115]. In extending the 4.4BSD scheduling infrastructure, we were careful to preserve its desirable properties, like non-starvation and priority decay.

The performance-aware policy is an extension of the scheduling infrastructure acting in a decoupled fashion. The priority of threads is adjusted using an additive term ($\Delta\text{priority}_{i,j}$) for each thread j of an SLO-bound application i. This operation induces the migration of the threads from a run queue to another, according to goals and performance. The 4.4BSD scheduler subdivides threads in five scheduling classes according to their assigned priority. The performance-aware policy works on threads in the *time-sharing user* class (*i.e.,* regular applications' threads) and further checks are applied in order to avoid the additive term to modify the scheduling class of a thread (*e.g.,* to the *real-time user* or to *idle* classes). The policy also sets an additional per-thread flag, allowing to realize the synergy with the thermal-aware policy; each thread j

of a performance-controlled application i is marked with either *force idle* or *prevent idle* as explained in Section 5.3.2.

The thermal-aware policy acts in coordination with the 4.4BSD scheduler: when the scheduler chooses the next thread to run, the control-theoretical system decides whether to actually execute it or to schedule the idle thread instead. The preemption of system critical threads (*i.e.,* bottom-half kernel threads, such as interrupt handlers, or top-half kernel threads) and of threads marked with *prevent idle* is automatically impeded, while the idle thread is always set to be executed if the next chosen thread by the 4.4BSD algorithm is marked with *force idle*.

5.5 EVALUATION

This section evaluates DPTM and, in particular, it is focused on answering the following question: can DPTM adapt CPU bandwidth allocation and, at the same time, perform idle cycle injection to allow an application within a multi-program workload to achieve its SLO, while constraining the operating temperature of the multicore processor?

We evaluated DPTM on a workstation with an Intel Core i7-870 Processor and 4 GB of main memory 2 memory modules. The processor features 4 cores operating at 2.97 GHz and sharing 8 MB of Last-Level Cache (LLC). Each of the memory module runs at 1066 MHz. We disabled the Enhanced Intel SpeedStep Technology and the Intel Turbo Boost Technology to prevent the processor entering P-states with clock frequency lower or higher than the nominal when a subset of the cores is executing. We disabled the Intel Hyper-Threading Technology (HTT) to simplify our implementation and analysis.

We configured FreeBSD 7.2 to boot with the kernel enhanced with all of our changes (*i.e.,* the performance monitoring infrastructure, the priority management scheme, and the idle cycle injection mechanism). The high-priority kernel threads for measuring the operating temperature of each core and implementing the idle cycle injection controller were set to run every 100 ms, which is the timeslice length of the 4.4BSD scheduler.

We assessed the behavior of DPTM through a multi-program workload combining 4 instances of *swaptions*, one of the benchmarks of the PARSEC 2.1 benchmark suite [50, 42].

When a single instance of *swaptions* runs freely on our evaluation platform, it peaks at 90000 Monte Carlo simulations/s. Conversely, when the 4 instances of the same application runs at the same time, they get a fair-share of the CPU bandwidth and achieves a throughput slightly higher than 20000 Monte Carlo simulations/s. The multi-program workload pushes the operating temperature of the CMP up to 80 °C. Our experiment consists in setting a performance requirements (*i.e.,* throughput) for an instance of *swaptions* and an operating temperature constraint. These values are above 40000 Monte Carlo simulations/s and below 60 °C, respectively.

Figure 42 shows both the throughput achieved by the SLO-bound instance of *swaptions* and the trace of the peak temperature across the CMP. This experiment provides evidence of the full capabilities of DPTM that, beyond applying preventive DTM to cap the peaks of the operating temperature, is also able to let SLO-bound applications to meet their performance requirements.

5.6 discussion and future work

The experimental evaluation of the DPTM framework, which we presented in this chapter describing its base methodology and its current implementation on FreeBSD 7.2, shows its efficacy in realizing preventive DTM while accounting for SLO-bound applications.

What is presented in this paper is a preliminary incarnation of the DPTM framework, aimed at demonstrating the soundness of the methodology; some simplifications were made in order to produce a proof-of-concept, opening ways for future works further improving the framework. For instance, the

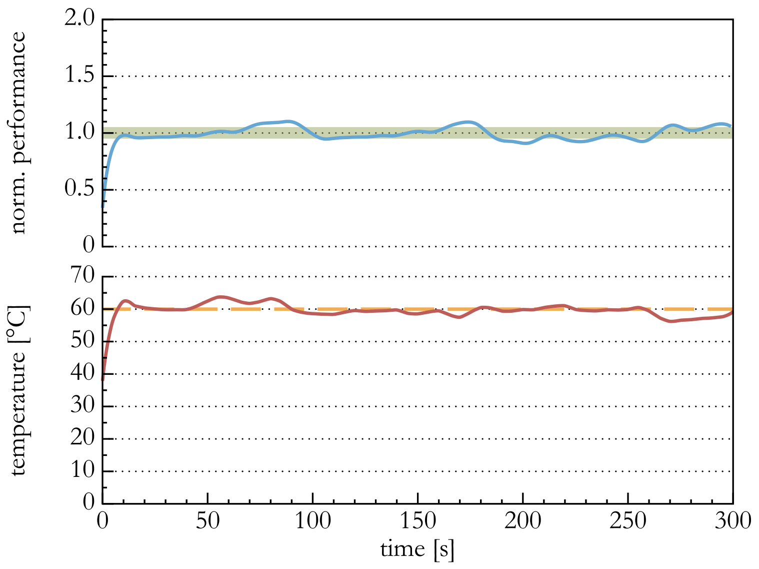


Figure 42.: Results with 4, 4-threaded instances of *swaptions*. One of the instances is SLO-bound and the operating temperature is constrained. Performance measurements are normalized to the SLO (*i.e.*, the performance requirement).

thermal-aware policy is currently based on per-core controllers considering only the measured core temperature for driving idle cycle injection; while this approach works in practice, it ignores thermal coupling among cores, which can be relevant on commodity CMP) and could be considered in future works. Another improvement could be refining the idle cycle injection mechanism to equally charge idle cycles to all the threads of a parallel application, avoiding artificial critical paths slowing down only one thread in case of synchronization points.

On the performance-aware policy side, an improvement would be also considering performance requirements in terms of latency or response time, which could allow characterizing applications such as web servers, making the DPTM framework more widely applicable. Moreover, the assumed performance model is not the most accurate possible and, despite working quite well in practice, the system could benefit from a more accurate modeling.

A topic non directly covered in this paper is the situation where SLOs cannot be met with available system resources. For instance, the experiment presented in Section 5.5 could be repeated by setting SLOs on all the four applications with performance requirements too high to be attained at the same time on the reference workstation. According to the current implementation of the DPTM framework, in this situation the thermal-aware policy would be disabled (*i.e.*, all the threads would be marked with *prevent idle*) and the available resources would be fairly distributed by the performance-aware policy to the SLO-bound applications weighted with respect to their goal. Hence, no application would respect its SLO, but the performance of each would be proportional to the respective requirements. This way, the system realizes sort of different QoS levels in case the system resources are not enough to fully respect the SLOs. Improvements in this scenario could come from refinements to the heuristic coupling the thermal and the performance-aware policies for more flexibility, allowing configuring different degrees of priority between the two.

6 CONCLUDING REMARKS

Back in 2010 we started the Autonomic Operating System (AcOS) research project to answer the question: can we enhance a commodity operating system (*e.g.,* the GNU's not Unix (GNU)/Linux operating system or the FreeBSD operating system) with an autonomic layer so as to achieve user-stated Service-Level Objective (SLO) expressed through high-level metrics and enforce administrator-stated constraints?

With the autonomic control loops (*i.e.,* designs implementing the Observe-Decide-Act (ODA) autonomic control loop) we proposed and validated in this dissertation, we contributed an affirmative answer. Metronome and Metronome++ demonstrated the ability to respect user-stated SLOs on performance measurements by means of Central Processing Unit (CPU) bandwidth and CPU core allocation, while ThermOS is capable of enforcing a system-level temperature constraints while still accounting for performance. Dynamic Performance and Thermal Management (DPTM) demonstrated that adaptive performance and thermal management is possible and functional even though we acknowledge that our design and implementation within the FreeBSD operating system is still preliminary. However, several open problems require further research before we can definitively answer this question.

An interesting direction is evaluating SLOs defined on different performance metrics (*e.g.,* latency, etc.). Resource allocation policies to automatically attain such requirements with the current state of on-chip shared resources [5] may need alternative resource allocation mechanisms [116].

Possibly, different SLO definitions may require the management of a wider set of resources (*e.g.,* cache and memory hierarchy, file system buffer cache, Input/Output (I/O) bandwidth, etc.). One of the challenges towards this direction is enabling mechanisms to effectively manage such resources at runtime. If coordinate management of multiple resources was demonstrated in a simulation environment [77], actual hardware and software mechanisms are needed to experiment with similar adaptation policies on commodity computing systems such as those provided by research operating systems [27, 117, 28].

Having an increasing pool of resources to manage and an increasing number of autonomic control loops leads to a third compelling challenge: properly orchestrating a large number of possibly conflicting adaptation policies. With DPTM [12], we propose an initial heuristic approach to define the interaction of two conflicting resource allocation policies aimed at achieving performance SLOs and enforcing a temperature constraint. A solution of this kind, however, does not scale well with an increasing number of autonomic control loops: we need to research a more systematic methodology [78].

We are confident that most of the solutions (*e.g.,* ThermOS) presented in this dissertation can scale on future Chip-Multiprocessor (CMP) with a higher number of CPU cores. However, an open question remains: how well the proposed solutions will scale at the datacenter level. Continue on this research line is of paramount important given the diffusion of warehouse-scale computing systems.

6.1 FUTURE WORK

Further development of some of the work presented in this dissertation is already on its way. First, we are moving from the operating system- to the hypervisor-level to address the adaptive performance management problem within cloud computing, which is possibly the more compelling research area for our contributions [118]. Second, we are dealing with additional resources such as the amount of shared Last-Level Cache (LLC) each Virtual Machine (VM) requires to achieve a user-stated SLO [116]. Third, we are moving towards distributed applications so as to take into account the possibly complex interactions among multiple VM executing on different physical machines [119, 120].

A CPU CORE ALLOCATION: A USER-SPACE APPROACH

In the latest decade, the information technology industry shifted from single to multicore processors. Multicore processors require better support from operating systems and runtimes to allow applications to achieve predictable performance and guarantee Quality of Service (QoS). Finding a proper schedule to yield the specified performance for single and multi-thread applications can be cumbersome; dealing with multi-program workloads may be even worse.

We present a performance-aware QoS-driven scheduler for multicore processors, which exploits the availability of runtime application-specific performance measurements to determine a suitable allotment of cores for multiprogrammed workloads so as to achieve the desired level of QoS. The proposed scheduler is meant to be implemented in user-space and harnesses an auto-regressive moving average performance model to put in a relationship performance measurements and resource allocation and is capable of embodying applications' characteristics such as execution phases.

A.1 INTRODUCTION

Multicore processors are ubiquitous in desktops, servers, and embedded devices. Computer architects designed multicore processors to overcome the limitations of superscalar processors (*e.g.,* poor performance per Watt ratios) whose performance stopped growing at historical rate.

This paradigm shift considerably increased the burden on systems' and applications' programmers. Nowadays, commodity operating systems schedulers fail in taking full advantage of multicore processors when scheduling multi-program workloads of multi-thread applications [58]. They are oblivious of applications' characteristics (*e.g.,* execution phases) and the resulting resource allocation may lead to unpredictable performance [121]. Judicious management of on-chip shared resources is critical to deliver predictable performance and (if possible) guarantee QoS.

Characterizing applications by means of their instructions' throughput [122], miss rate curves [123], speedup, efficiency (*i.e.,* speedup over resource allocation) [56], etc. to overcome the inefficiencies of commodity schedulers is a widely accepted practice [73, 58]. Characterization can be performance either offline, online, or by means of a mix of both. Profiling applications offline using reference (*i.e.,* training) inputs may not uncover execution phases that depends on the input itself. Moreover, anticipating a significant set of environmental conditions (*i.e.,* number of different multi-program workloads) is nearly prohibitive and, if possible, is likely to be time-consuming. As an alternative, it is possible collecting information at runtime to infer applications' characteristics and harnessing feedback-based mechanisms.

We understand and appreciate the value of offline analysis that can uncover fine-grained information; however, we advocate online characterization is key to guarantee QoS. Moreover, we cannot expect applications' programmers or even users to employ machine-specific performance measurements like Instructions Per Cycle (IPC) or Last-Level Cache (LLC) miss rate. Instead, we claim application-specific performance measurements (*e.g.,* frames/s for a video encoder or decoder) can be as effective as low-level once when harnessing feedback-based mechanisms. In addition, they are meaningful for applications' programmers and users since they address the impedance-mismatch problem [28] by turning the resource allocation problem into a goal definition problem, which is later bound to resource allocation.

To this end, this chapter presents a performance-aware QoS-driven scheduler for multicore processors running multi-program workloads and makes the following contributions:

- Introducing a simple yet effective user-mode performance monitoring infrastructure to instrument applications so as to make application-specific performance measurements system-wide accessible and allow users specifying QoS requirements;
- Exploiting a first order discrete-time Auto-Regressive Moving Average (ARMA) performance model to establish the relationship between resource allocation and performance measurements;
- Harnessing online estimation to discover the performance model's parameters through a recursive least square (RLS) filter, thus uncovering applications' characteristics such as coarse-grain execution phases [64];
- Implementing a performance manager leveraging a proportional-integral (PI) controller feeding resource demands to a fair resource allocator, which exploits well-established resource allocation mechanisms.

In the rest of this chapter: Appendix A.2 illustrates the design principles and gives development insights. Appendix A.3 presents an experimental campaign showing the validity of the proposed approach. Appendix A.4 goes through the related works and, finally, Appendix A.5 concludes the chapter.

A.2 DESIGN AND DEVELOPMENT

The scheduler proposed in this chapter leverages a classic feedback-based structure consisting of three distinct phases respectively devoted to:

1. Monitor applications to gather performance measurements;
2. Evaluate the scheduler's policy devising a thread to core mapping so as to guarantee (if possible) QoS;
3. Apply the mapping migrating threads as needed.

These three phases construct a closed loop as depicted in Figure 43.

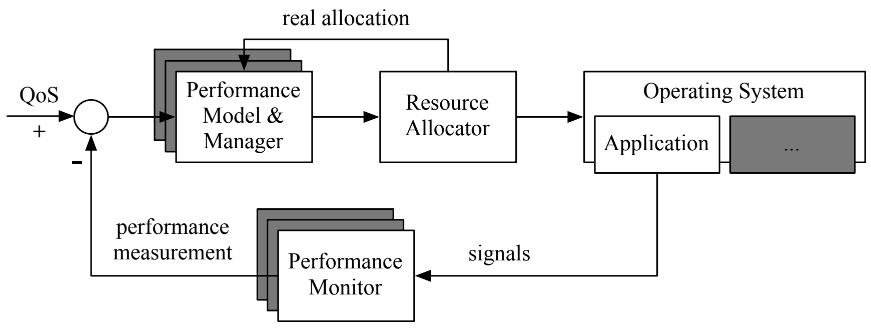


Figure 43.: System architecture diagram. Each application is coupled with an instance of the performance monitor and one of the performance model and manger. A single *resource allocator* normalizes applications' resource demands.

A.2.1 *Performance Monitoring*

The Heart Rate Monitor (HRM) [8] is an open source monitoring infrastructure that compute application-specific performance measurements on which users can specify QoS requirements through applications' instrumentation. Adaptation policies, which can be implemented in kernel [8, 9] and user-space [51], can exploit the availability of performance measurements and QoS requirements to affect applications' execution. HRM is tightly integrated with the Linux kernel since its primary goal was to export application-specific performance measurements to the kernel-space.

This chapter proposes a more general approach leveraging a user-space scheduler that can run on top of most Portable Operating System Interface for Unix (POSIX)-compliant operating systems (*e.g.,* GNU's not Unix (GNU)/Linux, FreeBSD, ...) with minimal changes. Due to this reason, we developed a portable *user-space* performance monitoring infrastructure: *libthroughput* with performance comparable to that of HRM. *libthroughput* delivers competitive performance with respect to HRM and provides similar functionality: first, performance measurements for both single and multi-threaded applications and multi-programmed applications, second, QoS requirements specification.

libthroughput collects application-specific performance measurements representing applications' throughput. As an example, consider the *x264* video encoder, which implements the H.264/Moving Picture Experts Group (MPEG)-4 Part 10 or Advanced Video Coding (AVC) standard, included in the PARSEC 2.1 benchmark suite [42]. The parallel algorithm of *x264* harnesses a virtual pipeline with one stage per frame. *x264* processes in parallel a number of pipeline stages equal to the number of encoder threads realizing a sliding window moving from the beginning to the end of the pipeline. Once the execution of a stage finishes the encoder thread handling the stage issues a *signal*, which in the context of *x264* signifies the completion of a frame. Users can specify QoS requirements through a meaningful performance measurement like frames/s.

A.2.2 *Performance Modeling and Management*

The proposed user-space scheduler leverages a first order discrete-time ARMA *performance model* that established the relationship between the resource allocation and performance measurements. Equation (26) reports the mathematical formulation where $r(k)$ and $r(k+1)$ are the performance measurements at the k-th and $(k+1)$-th control steps, respectively. $c(k)$ is the subset of cores allocated to the application. a and b are the performance model's parameters whose values depend on the application, the workload, and the system.

$$r(k+1) = a \cdot r(k) + b \cdot c(k) \tag{26}$$

The proposed user-space scheduler employs a Recursive Least Squares (RLS) filter, which is a common choices among least squares and Kalman filters [77], to perform online estimation of the performance model's parameters. Online estimation allows the user-space scheduler to capture applications' characteristics such as execution phases, which changes the relationship between the resource allocation and performance measurements. Appendix A.3.1 reports a thorough validation of the performance model.

Starting from the performance model reported in Equation (26) we devised a *performance manager* leveraging a PI controller that responds to errors (*i.e.,* difference between the QoS requirements and performance measurements) by means of two terms: a proportional and an integral term. The proportional term changes its effect according to the current value of the error and in a way that reduces the future values of the error. The integral term changes its effect incrementally accounting for the past values of the error. The choice of neglecting the derivative term, thus the use of a proportional-integral-derivative (PID) controller, translates into a little loss of control but, at the same time, leads to notable less noise.

Equation (27) reports the mathematical formulation of the PI controller for application i.

$$c_i(k) = c_i(k-1) + \frac{1-p}{b} \cdot e_i(k) - a \cdot \frac{1-p}{b} \cdot e_i(k-1) \tag{27}$$

$c_i(k)$ and $c_i(k-1)$ are the subset of cores required by the application between the k-th and the $(k+1)$-th control steps to respect the QoS requirement and the subset of cores allocated to the application between the $(k-1)$-th and the k-th control steps, respectively. $e_i(k)$ and $e_i(k-1)$ are the errors at the k-th and $(k-1)$-th control steps, respectively; the errors are computed as $\bar{r_i} - r_i(k)$ and $\bar{r_i} - r_i(k-1)$, where $\bar{r_i}$ is the QoS requirement.

We synthesized the PI controller by applying classical control theory techniques as explained by Hellerstein et al. [96] and constrained the transfer function to have a single pole in p. The controller's parameter p affects the responsiveness; if the value is chosen in the interval $(-1, 1)$ the system is guaranteed to be stable as long as the performance model holds.[1] Furthermore, if the value is chosen in the interval $(0, 1)$ the system is guaranteed to avoid oscillations. In

1 The use of adaptive control through the RLS filter enforces the performance model.

general, large values in the interval $(0, 1)$ translate into a slower but smoother response, while small values translate into a faster but rougher response.

A.2.3 *Resource Allocation*

Each *performance manager* i computes the subset of cores the application i requires to satisfy the QoS requirement. These computations are carried on independently, thus resulting in either a demand that can or cannot be satisfied by the number of cores available in the system.

Whenever the demand can be satisfied the user-space scheduler can exploit an energy-aware policy shutting down unused cores through clock and power gating.

On the other hand, if the demand cannot be satisfied, the *resource allocator* can harness many different policies. We propose re-distributing cores according to *performance managers'* demands following a performance-aware fair policy similar to one we proposed with Metronome [8]. Equation (28) reports the mathematical formulation of the re-distributing filter where $\bar{c}$ is number of cores available in the system and $\tilde{c}_i(k)$ is the proportional demand for application i at the k-th control step.

$$\tilde{c}_i(k) = c_i(k) \cdot \frac{\bar{c}}{\sum_j c_j(k)} \tag{28}$$

The *resource allocator* is also in charge of rounding the floating-point number of cores as needed and inform *performance managers* to avoid compromising the auto-tuning process. Moreover, the *resource allocator* ensures that applications receive at least one core at every control step maintaining the highly desirable non-starvation property of most commodity schedulers.

Alternative policies can employ weights to provide additional knobs and different service levels for different users.

A.2.4 *Prototype for GNU/Linux*

The overall design of the user-space scheduler is general enough to be implemented on top of most POSIX-compliant operating systems. Given our design, we developed a prototype on top of the GNU/Linux operating system and its commodity scheduler.

Both performance monitoring and resource allocation heavily relies on the infrastructure provided by GNU/Linux. *libthroughput* exploits *cgroups* [55] to inform the *resource allocator* about which threads belong to which application by creating a new cgroup for each application. The *resource allocator* maps threads belonging to an application to a subsets of cores by harnessing the *cpuset* subsystem of cgroups.

A.3 EVALUATION

We run all the experiments to evaluate the effectiveness of the performance-aware QoS-driven scheduler on a Dell Precision T3500 server with an Intel Xeon Processor W3670 and 12 GB of main memory in 3 memory modules. The processor features 6 cores clocked at 3.20 GHz and sharing 12 MB of LLC. Each memory module features chips clocked at 1066 MHz. We disabled the Enhanced Intel SpeedStep Technology, the Intel Turbo Boost Technology, and the Intel Hyper-Threading Technology (HTT) to simplify our analysis. The operating system is Debian GNU/Linux 7.0 x86-64 with the Linux kernel 3.2.

A.3.1 *Performance Model Validation*

We evaluate the effectiveness of the performance model against a subset of the applications from the PARSEC 2.1 benchmark suite [50, 42] instrumented with *libthroughput*. We run each application 100 times randomly varying the

Table 10.: Performance model assessment through the average and the standard deviation of the coefficient of determination and the mean absolute percentage error over six static resource allocations

application	R^2 [0, 1]		MAPE [%]	
	avg.	std dev.	avg.	std dev.
blackscholes	0.97	0.01	0.63	0.04
bodytrack	0.86	0.02	1.10	0.05
canneal	0.96	0.01	0.95	0.04
dedup	0.73	0.02	3.38	0.17
facesim	0.89	0.02	1.03	0.04
ferret	0.92	0.02	0.83	0.03
swaptions	0.98	0.01	0.51	0.02
x264	0.72	0.03	5.48	0.26

subset of cores allocated and collecting performance measurements through *libthroughput*. We applied the least squares algorithm to regress the performance model's parameters.

We then run each application fixing the subset of cores allocated (from 1 to 6) and computed the coefficient of determination R^2 and the Mean Absolute Percentage Error (MAPE) as suggested by Sharifi et al. [77] using the application-specific performance measurements provided by *libthroughput* instead of the IPC.

Table 10 reports the average and the standard deviation of R^2 and MAPE for the applications we analyzed. The averages of the first metric, R^2, is close to 1, which means the performance model is quite accurate; the small standard deviations say the averages holds for most of the applications apart from *dedup* and *x264*. These two applications go through different execution phases, thus benefiting from the online estimation of the performance model's parameters that is not employed for performance model assessment. The averages and standard deviations of the second metric, MAPE, leads to similar considerations with the addition of quantitative information on the percentage error.

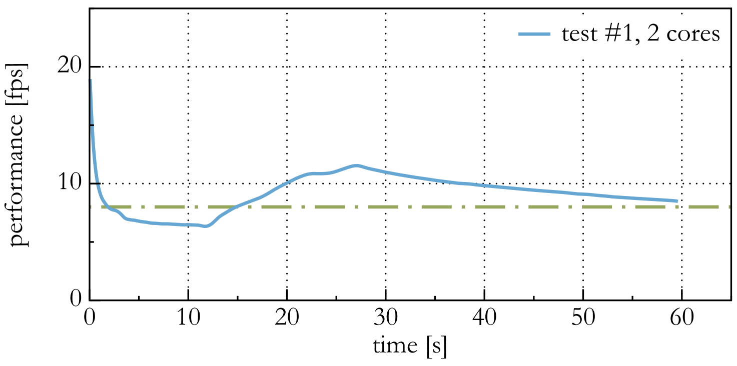
The rest of this section focuses on the experiments with two instances of *x264*. Among the subset of the PARSEC 2.1 benchmark suite we employed, *x264* is the most challenging application and provides the opportunity to evaluate complex scenarios, especially when two instances run at the same time.[2]

A.3.2 *Static Resource Allocation: Comparison*

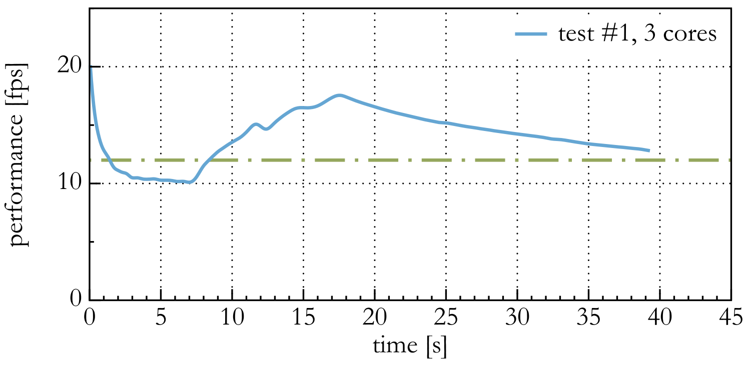
We compare the proposed user-space scheduler with two standard (static) resource allocation mechanisms provided by the cgroups subsystem of the Linux kernel: cpuset and bandwidth. cpuset allows a group of threads to use a subset of the available cores. The cpuset subsystem is an example of spatial scheduling solution while the bandwidth is a more classical time-share scheduling approach. The bandwidth subsystem enforces the reservation of a certain quota of the multicore processor computational power over a period of time for a cgroup.

Figures 44a and 44b show the behavior (see the blue lines in the figures) of the first and second instances of *x264*, respectively, with the best static resource allocation to respect the QoS requirements of 8 and 12 frames/s or fps through the cpuset subsystem (see the green dash-dot lines in the figures). The first instance is assigned 2 cores (*i.e.,* 0 and 1) while the second instance gets 3 cores (*i.e.,* 2-4). With these static resource allocations both the applications satisfy the QoS requirements at the end of the execution; however, during the execution the performance measurements vary a lot between 6 and 13 fps for the first instance and between 10 and 18 fps for the second instance. This is due to the different execution phases, which for *x264* are input-dependent, the application goes through. Ideally, one would want to keep the performance measurements as close as possible to the QoS requirements during the whole execution to avoid wasting resources. The take out of this experiment is that: there exists no static resource allocation that can be achieved through the cpuset subsystem

2 We show only the experimental results obtained with *x264* for space constraints. Moreover, we limited our study to workloads made up of two applications since our evaluation platform cannot afford running more applications with reasonable performance.



(a) Performance profile of the first instance running with 2 cores.



(b) Performance profile of the second instance running with 3 cores.

Figure 44.: Performance profiles of the two instances of *x264* running simultaneously with static resource allocations obtained through the cpuset subsystem.

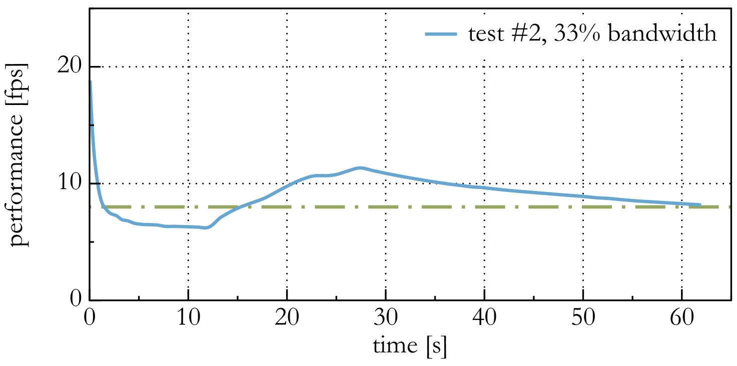
such that both the instances of *x264* respect the QoS requirements during the whole execution.

Figures 45a and 45b display the experimental results obtained by replicating the previous experiments exploiting the bandwidth subsystem. With the bandwidth subsystem, applications make use of all the cores available, which are six on our evaluation platform. The first instance of *x264* is assigned 33 % of the bandwidth of the multicore processors, which is approximately equal to the 2 cores assigned in the previous experiment. The second instance gets 47 % of the bandwidth of the multicore processors, which is, once again, approximately equal to the 3 cores assigned in the previous experiment. Even though the bandwidth subsystem is much finer-grained than the cpuset subsystem, the take out of this experiment is the same of the first. It is worth noting that coupling the cpuset and bandwidth subsystems to perform static resource allocation yields equally unsatisfactory experimental results.

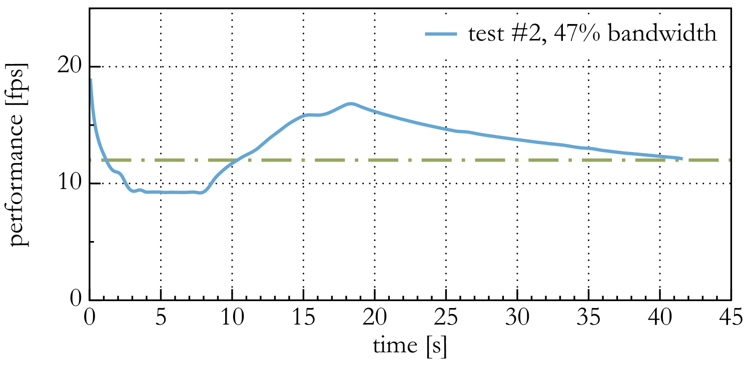
A.3.3 *Dynamic Resource Allocation*

We evaluate the proposed performance-aware QoS-driven scheduler for multicore processors in the same scenario of the static resource allocation, with the same multi-program workload and the same QoS requirements. *Performance managers* run every 50 ms and update the performance model at the same frequency by retrieving performance measurements through *libthroughput*, which is capable of computing fresh information every 10 ms. *Performance managers* demand new resource allocations and coordinate through the *resource allocator* every 500 ms. Each of these periods is configurable and different configuration benefit different applications depending on the execution phases they may go through.

Figures 46a and 46b show the experimental results obtained by running the two instances of *x264* with QoS requirements of 8 and 12 fps, respectively. Both the performance profiles track the QoS requirements after an initial settling phase in which the performance model's parameters converge to their "real"

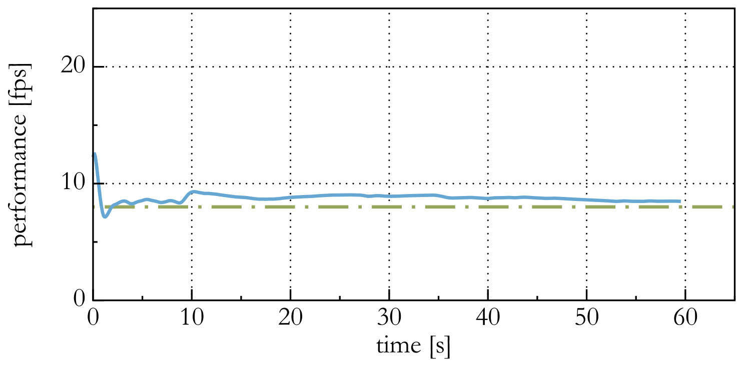


(a) Performance profile of the first instance running with 33 % of the bandwidth.

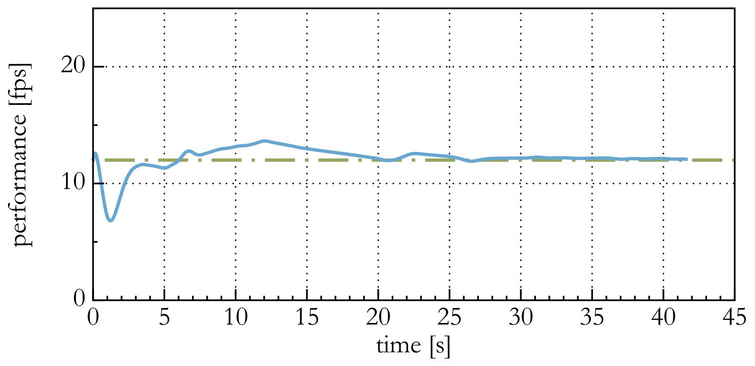


(b) Performance profile of the second instance running with 47 % of the bandwidth.

Figure 45.: Performance profiles of the two instances of *x264* running simultaneously with static resource allocations obtained through the bandwidth subsystem.



(a) Performance profile of the first instance meeting the 8 fps requirement.



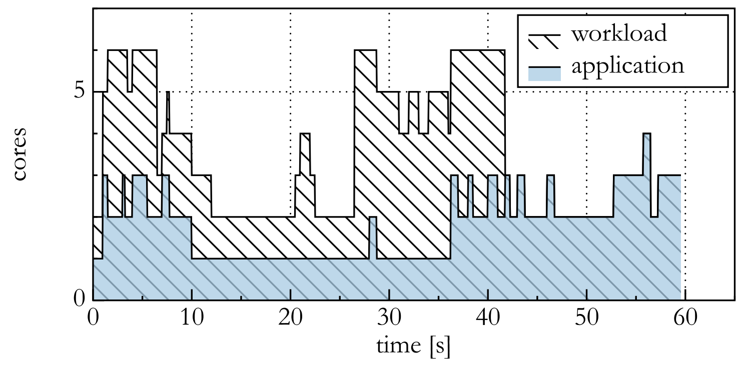
(b) Performance profile of the second instance meeting the 12 fps requirement.

Figure 46.: Performance profiles of the two instances of *x264* running simultaneously with the proposed user-space scheduler.

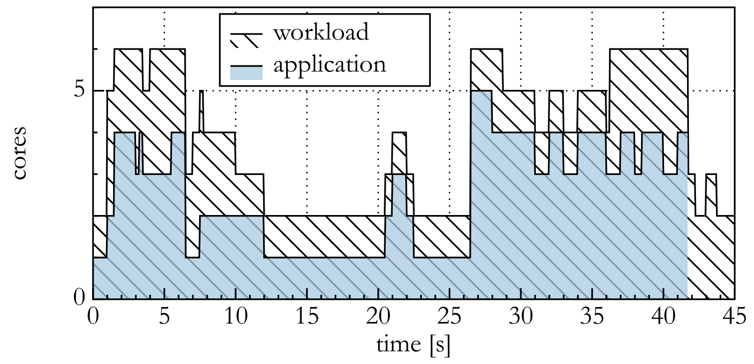
values. Figures 47a and 47b display the subset of the available cores assigned to the first and to the second instance of *x264*, respectively. The resource allocations follow the performance profile of the application, decreasing the number of cores when the performance measurements naturally rise because of the lower complexity of the video and increasing the number of cores when the performance measurements fall due to the higher complexity of the video. Moreover, it is important to notice how the sum of the number of cores assigned to the first and second instances of *x264* never exceeds number of available cores thanks to the re-distributing filter inside the *resource allocator*.

A.3.4 *Discussion*

The choice of exploiting the cpuset subsystem instead of the bandwidth subsystem is actually sub-optimal for the proposed approach for two reasons: first, the cpuset subsystem is coarser-grained than the bandwidth subsystem and, second, the cpuset subsystem requires, like the bandwidth subsystem, running each multi-thread application with a number of threads which is at least as high as the number of available cores. The first drawback is easy to understand since the cpuset subsystem allows partitioning the bandwidth of the hexa-core processors by multiple of $\approx$ 16 %. This issue is simply addressed by increasing the decision/actuation frequency of the *performance managers* and *resource allocator*. The cpuset subsystem enables resource allocation on space axis; variable dynamic resource allocation either requires multi-thread applications to vary the number of threads accordingly or to run with a number of threads that allows exploiting the full parallelism of the multicore processor. Conversely, the bandwidth subsystem always requires multi-thread applications to run with an adequate number of threads since resource allocation is performed on the time axis. With the cpuset subsystem multi-thread applications may end in unbalanced configurations where some cores must handle more threads than others possibly introducing artificial critical paths due to synchronization



(a) Core allocation for the first instance while meeting the 8 fps requirement.



(b) Core allocation for the second instance while meeting the 12 fps requirement.

Figure 47.: Core allocations for the two instances of *x264* running simultaneously with the proposed user-space scheduler.

issues. This is an issue we observe with the proposed approach and it may require the adoption of advance load balancing scheme such as Juggle [124].

A.4 RELATED WORK

Recently there has been extensive research on solutions to maximize performance and/or respect QoS requirements.

Researchers focused on cache partitioning approaches [66, 68], memory bandwidth partitioning solutions [125], cores partitioning algorithms [73, 58], and both time and space-sharing approaches [8, 9]. Moreover, these works exploited different decision-making techniques spanning from heuristics [73, 8, 58, 9] to machine learning [75, 51], control theory [77], a mix of them [78].

The proposed approach borrows the design of HRM from Metronome [8] to leverage application-specific performance measurements and explores the use of a PI controller instead of the speedup-based approach from Metronome++ [9]. Like *METE* [77], our user-space scheduler employs an ARMA performance model coupled with a PI controller to implement the *performance manager*. *METE* handles multiple resources (*i.e.,* cores, cache ways, and memory bandwidth) while the proposed approach partitions a single resource (*i.e.,* cores); our limitation is due to the fact that we implemented the proposed approach on real hardware, while *METE* cannot be implemented since cache ways and memory bandwidth partitioning is not supported by any commodity multicore processor.

Orthogonal approaches [81] dynamically adjust the number of threads within multi-threaded applications to optimize the overall efficiency of the system or proactively addresses the load balancing issue [124]. Coupling the proposed approach with these approaches may solve the second issue discussed in Appendix A.3.4. The availability of Scheduler Activations [126] can improve the efficiency of the proposed approach avoiding costly kernel-space thread migrations in favor of user-space thread migrations.

The proposed approach might recall a real-time scheduling infrastructure. However, we believe the two scheduling solutions are different in some of the key aspects. Let us focus on priority-based real-time scheduling infrastructures that are the one resembling more the proposed scheduling approach. The Earliest Deadline First (EDF) is a priority-based scheduling algorithm; with EDF an application i specifies a relative deadline D_i and a worst-case execution time C_i. Applications programmers may need to overestimate C_i to account of workload variations among the deadlines. The overestimation may in turn lead to a waste of resources (*e.g.,* the scheduling infrastructure implements a hard admission control policy and no application can exploit unused resources). The proposed approach borrows ideas from adaptive systems and accounts for application-specific performance measurements and QoS requirements to gracefully adapt resource allocation shifting from resource-centric solutions like real-time scheduling infrastructures to a goal-oriented one.

A.5 CONCLUSIONS

We presented a performance-aware QoS-driven scheduler for multicore processors and multi-program workloads that sits on top of well-established resource allocation mechanisms, namely the cgroups subsystem of the Linux kernel. The proposed approach harnesses application-specific performance measurements and QoS requirements provided through *libthroughput*, the user-space dual of HRM to address the impedance-mismatch problem and turn the resource allocation problem into a goal-definition problem. In addition, the proposed approach leverages a discrete-time ARMA performance model and a RLS filter to dynamically establish the relationship between the resource allocation and the performance measurements. A set of PI controllers determine suitable allotments of cores so as applications can respect (if possible) QoS requirements. Experimental results on a commodity multicore processor with emerging real-world applications highlight the effectiveness of the proposed approach that allows applications respecting QoS requirements even in presence of execution phases.

BIBLIOGRAPHY

[1] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974. 10.1109/JSSC.1974.1050511.

[2] John Markoff. The not-so-distant future of personal computing. *InfoWorld*, 49, 1993.

[3] Gordon E. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1), January 1998. 10.1109/JPROC.1998.658762.

[4] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, OSDI '10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[5] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th International Symposium on Microarchitecture*, MICRO '11, pages 248–259, New York, NY, USA, 2011. ACM. 10.1145/2155620.2155650.

[6] Hiroshi Sasaki, Satoshi Imamura, and Koji Inoue. Coordinated Power-Performance Optimization in Manycores. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 51–62, Piscataway, NJ, USA, 2013. IEEE Press.

[7] Parthasarathy Ranganathan. Recipe for Efficiency: Principles of Power-Aware Computing. *Commun. ACM*, 53(4):60–67, April 2010. 10.1145/1721654.1721673.

[8] Filippo Sironi, Davide B. Bartolini, Simone Campanoni, Fabio Cancare, Henry Hoffmann, Donatella Sciuto, and Marco D. Santambrogio. Metronome: Operating System Level Performance Management via Self-Adaptive Computing. In *Proceedings of the 49th Design Automation Conference*, DAC '12, pages 856–865, New York, NY, USA, 2012. ACM. 10.1145/2228360.2228514.

[9] Davide B. Bartolini, Riccardo Cattaneo, Gianluca C. Durelli, Martina Maggio, Marco D. Santambrogio, and Filippo Sironi. The Autonomic Operating System Research Project: Achievements and Future Directions. In *Proceedings of the 50th Design Automation Conference*, DAC '13, pages 77:1–77:10, New York, NY, USA, 2013. ACM. 10.1145/2463209.2488828.

[10] Filippo Sironi, Donatella Sciuto, and Marco D. Santambrogio. A Performance-Aware Quality of Service-Driven Scheduler for Multicore Processors. *SIGBED Rev.*, January 2014.

[11] Filippo Sironi, Martina Maggio, Riccardo Cattaneo, Giovanni Francesco Del Nero, Donatella Sciuto, and Marco Domenico Santambrogio. ThermOS: System Support for Dynamic Thermal Management of Chip Multiprocessors. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 41–50, Piscataway, NJ, USA, 2013. IEEE Press. 10.1109/PACT.2013.6618802.

[12] Davide B. Bartolini, Filippo Sironi, Martina Maggio, Riccardo Cattaneo, Donatella Sciuto, and Marco D. Santambrogio. A Framework for Thermal and Performance Management. In *Proceedings of the Workshop on Managing Systems Automatically and Dynamically*, MAD '12, Berkeley, CA, USA, 2012. USENIX Association.

[13] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, January 2003. 10.1109/MC.2003.1160055.

[14] Simon Dobson, Roy Sterritt, Paddy Nixon, and Mike Hinchey. Fulfilling the Vision of Autonomic Computing. *IEEE Computer*, 43(1):35–41, January 2010. 10.1109/MC.2010.14.

[15] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2), May 2009. 10.1145/1516533.1516538.

[16] Markus C. Huebscher and Julie A. McCann. A Survey of Autonomic Computing—Degrees, Models, and Applications. *ACM Comput. Surv.*, 40(3), August 2008. 10.1145/1380584.1380585.

[17] Peter Bailis, Vijay Janapa Reddi, Sanjay Gandhi, David Brooks, and Margo Seltzer. Dimetrodon: Processor-level Preventive Thermal Management via Idle Cycle Injection. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 89–94, New York, NY, USA, 2011. ACM. 10.1145/2024724.2024745.

[18] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th European Conference on Computer Systems*, EuroSys '09, pages 89–102, New York, NY, USA, 2009. ACM. 10.1145/1519065.1519076.

[19] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and The Art of Virtualization. In *Proceedings of the 19th Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM. 10.1145/945445.945462.

[20] Samuel H. Fuller and Lynette I. Millett. Computing Performance: Game Over or Next Level? *IEEE Computer*, 44(1):31–38, January 2011. 10.1109/MC.2011.15.

[21] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 79–88, New York, NY, USA, 2010. ACM. 10.1145/1809049.1809065.

[22] Brinkley Sprunt. The Basics of Performance-Monitoring Hardware. *IEEE Micro*, 22(4):64–71, July 2002. 10.1109/MM.2002.1028477.

[23] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, August 2000. 10.1177/109434200001400303.

[24] Innovative Computing Laboratory (University of Tennessee). Performance Application Programming Interface. Retrieved Sept. 1, 2013 from `http://icl.cs.utk.edu/papi/`, 2013.

[25] Brinkley Sprunt. Managing The Complexity Of Performance Monitoring Hardware: The Brink and Abyss Approach. *Int. J. High Perform. Comput. Appl.*, 20(4):533–540, November 2006. 10.1177/1094342006064569.

[26] Calin Cascaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. Performance and Environment Monitoring for Continuous Program Optimization. *IBM J. Res. Develop.*, 50(2.3):239–248, 2006. 10.1147/rd.502.0239.

[27] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a Complete Operating System. In *Proceedings of the 1st European Conference on Computer Systems*, EuroSys '06, pages 133–145, New York, NY, USA, 2006. ACM. 10.1145/1217935.1217949.

[28] Juan A. Colmenares, Gage Eads, Steven Hofmeyr, Sarah Bird, Miquel Moretó, David Chou, Brian Gluzman, Eric Roman, Davide B. Bartolini,

Nitesh Mor, Krste Asanović, and John D. Kubiatowicz. Tessellation: Refactoring the OS Around Explicit Resource Containers with Continuous Adaptation. In *Proceedings of the 50th Design Automation Conference*, DAC '13, New York, NY, USA, 2013. ACM. 10.1145/2463209.2488827.

[29] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. Tessellation: Space-time Partitioning in a Manycore Client OS. In *Proceedings of the 1st Workshop on Hot Topics in Parallelism*, HotPar '09, Berkeley, CA, USA, 2009. USENIX Association.

[30] Juan A. Colmenares, Sarah Bird, Henry Cook, Paul Pearce, David Zhu, John Shalfy, Steven Hofmeyry, Krste Asanović, and John Kubiatowicz. Resource Management in the Tessellation Manycore OS. In *Proceedings of the 2nd Workshop on Hot Topics in Parallelism*, HotPar '10, Berkeley, CA, USA, 2010. USENIX Association.

[31] Roberto A. Vitillo. perf: Linux profiling with performance counters. Retrieved Sept. 1, 2013 from `https://perf.wiki.kernel.org/index.php/Main_Page`, 2013.

[32] Intel Corp. Intel Performance Counter Monitor – A better way to measure CPU utilization. Retrieved Sept. 1, 2013 from `http://goo.gl/s5x7RA`, August 2012.

[33] Alaa R. Alameldeen and David A. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, July 2006. 10.1109/MM.2006.73.

[34] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, pages 119–130, 2012.

[35] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, OSDI '10, pages 33–46, Berkeley, CA, USA, 2010. USENIX Association.

[36] Livio Soares and Michael Stumm. Exception-Less System Calls for Event-Driven Servers. In *Proceedings of the Annual Technical Conference*, ATC '11, pages 131–144, Berkeley, CA, USA, 2011. USENIX Association.

[37] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[38] The Apache Software Foundation. The Apache HTTP Server Project. Retrieved Sept. 1, 2013 from `http://httpd.apache.org/`, 2013.

[39] Linux-HA. The Linux High Availability Project. Retrieved Sept. 1, 2013 from `http://www.linux-ha.org/wiki/Main_Page`, July 2009.

[40] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, OSDI '04, pages 137–149, Berkeley, CA, USA, 2004. USENIX Association.

[41] VideoLAN Organization. x264, the best H.264/AVC encoder. Retrieved Sept. 1, 2013 from `http://www.videolan.org/developers/x264.html`, 2013.

[42] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[43] Matteo Carminati. *Self-Aware Synchronization Mechanisms and Decision Making Techniques for Contention-Aware Thread Mapping*. Master's thesis, University of Illinois at Chicago, May 2012.

[44] Standard Performance Evaluation Corp. SPECjbb2013. Retrieved Sept. 1, 2013 from `http://www.spec.org/jbb2013/`, 2013.

[45] Standard Performance Evaluation Corp. SPECweb2009. Retrieved Sept. 1, 2013 from `http://www.spec.org/web2009/`, 2013.

[46] Transaction Processing Performance Council. TPC - Benchmarks. Retrieved Sept. 1, 2013 from `http://www.tpc.org/information/benchmarks.asp`, 2013.

[47] Standard Performance Evaluation Corp. SPEC CPU2006. Retrieved Sept. 1, 2013 from `http://www.spec.org/cpu2006/`, 2013.

[48] Standard Performance Evaluation Corp. SPEC OMP2012. Retrieved Sept. 1, 2013 from `http://www.spec.org/omp2012/`, 2013.

[49] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM. 10.1145/223982.223990.

[50] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM. 10.1145/1454115.1454128.

[51] Jacopo Panerati, Filippo Sironi, Matteo Carminati, Martina Maggio, Giovanni Beltrame, Piotr J. Gmytrasiewicz, Donatella Sciuto, and Marco D. Santambrogio. On Self-adaptive Resource Allocation through Reinforcement Learning. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems*, AHS '13, pages 23–30, Piscataway, NJ, USA, 2013. IEEE Press. 10.1109/AHS.2013.6604222.

[52] Kenneth C. Knowlton. A Fast Storage Allocator. *Commun. ACM*, 8(10): 623–624, October 1965. 10.1145/365628.365655.

[53] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic Knobs for Responsive Power-aware Computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 199–212, New York, NY, USA, 2011. ACM. 10.1145/1950365.1950390.

[54] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore

Scaling. In *Proceedings of the 38th International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM. 10.1145/2000064.2000108.

[55] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association.

[56] D. L. Eager, J. Zahorjan, and E. D. Lozowska. Speedup Versus Efficiency in Parallel Systems. *IEEE Trans. Comput.*, 38(3):408–423, March 1989. 10.1109/12.21127.

[57] Wonyoung Kim, Meeta Sharma Gupta, Gu-Yeon Wei, and David Brooks. System Level Analysis of Fast, Per-Core DVFS using On-Chip Switching Regulators. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture*, HPCA '08, pages 123–134, Washington, DC, USA, 2008. IEEE Computer Society. 10.1109/HPCA.2008.4658633.

[58] Hiroshi Sasaki, Teruo Tanimoto, Koji Inoue, and Hiroshi Nakamura. Scalability-Based Manycore Partitioning. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 107–116, New York, NY, USA, 2012. ACM. 10.1145/2370816.2370833.

[59] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[60] Juri Lelli, Giuseppe Lipari, Dario Faggioli, and Tommaso Cucinotta. An efficient and scalable implementation of global EDF in Linux. In *Proceedings of the 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, OSPERT '11, 2011.

[61] M. Tim Jones. Inside the Linux 2.6 Completely Fair Scheduler – Providing fair access to CPUs since 2.6.23. Retrieved

Sept. 1, 2013 from `http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/`, Design Automation Conference 2009.

[62] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-efficiency While Preserving Responsiveness. In *Proceedings of the 40th International Symposium on Computer Architecture*, ISCA '13, pages 308–319, New York, NY, USA, 2013. ACM. 10.1145/2485922.2485949.

[63] Filippo Sironi. $(\mathrm{PA})^2$: a Scheme for Performance-Aware Processor Allocation for Chip-Multiprocessors. Technical report, Politecnico di Milano, 2012.

[64] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 45–57, New York, NY, USA, 2002. ACM. 10.1145/605397.605403.

[65] Jichuan Chang and Gurindar S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *Proceedings of the 21st International COnference on Supercomputing*, ICS '07, pages 242–252, New York, NY, USA, 2007. ACM. 10.1145/1274971.1275005.

[66] Mahmut Kandemir, Taylan Yemliha, and Emre Kultursay. A Helper Thread Based Dynamic Cache Partitioning Scheme for Multithreaded Applications. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 954–959, New York, NY, USA, 2011. ACM. 10.1145/2024724.2024936.

[67] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and Efficient Fine-grain Cache Partitioning. In *Proceedings of the 38th International Symposium on Computer Architecture*, ISCA '11, pages 57–68, New York, NY, USA, 2011. ACM. 10.1145/2000064.2000073.

[68] Akbar Sharifi, Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. Courteous Cache Sharing: Being Nice to Others in Capacity Management. In *Proceedings of the 49th Design Automation Conference*, DAC '12, pages 678–687, New York, NY, USA, 2012. ACM. 10.1145/2228360.2228482.

[69] Nathan Beckmann and Daniel Sanchez. Jigsaw: Scalable Software-defined Caches. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 213–224, Piscataway, NJ, USA, 2013. IEEE Press.

[70] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, PACT '07, pages 245–258, Washington, DC, USA, 2007. IEEE Computer Society. 10.1109/PACT.2007.29.

[71] Onur Mutlu and Thomas Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proceedings of the 40th International Symposium on Microarchitecture*, MICRO '07, pages 146–160, Washington, DC, USA, 2007. IEEE Computer Society. 10.1109/MICRO.2007.40.

[72] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *Proceedings of the 35th International Symposium on Computer Architecture*, ISCA '08, pages 39–50, Washington, DC, USA, 2008. IEEE Computer Society. 10.1109/ISCA.2008.21.

[73] Julita Corbalán, Xavier Martorell, and Jesús Labarta. Performance-Driven Processor Allocation. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, OSDI '00, Berkeley, CA, USA, 2000. USENIX Association.

[74] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In *Proceedings of the 16th International Conference on Parallel*

Architectures and Compilation Techniques, PACT '07, pages 25–38, Washington, DC, USA, 2007. IEEE Computer Society. 10.1109/PACT.2007.40.

[75] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach. In *Proceedings of the 41st International Symposium on Microarchitecture*, MICRO '08, pages 318–329, Washington, DC, USA, 2008. IEEE Computer Society. 10.1109/MICRO.2008.4771801.

[76] Shekhar Srikantaiah, Reetuparna Das, Asit K. Mishra, Chita R. Das, and Mahmut Kandemir. A Case for Integrated Processor-cache Partitioning in Chip Multiprocessors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, 2009. ACM. 10.1145/1654059.1654066.

[77] Akbar Sharifi, Shekhar Srikantaiah, Asit K. Mishra, Mahmut Kandemir, and Chita R. Das. METE: Meeting End-to-end QoS in Multicores Through System-wide Resource Management. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '11, pages 13–24, New York, NY, USA, 2011. ACM. 10.1145/1993744.1993747.

[78] Henry Hoffman. *SEEC: A Framework for Self-aware Management of Goals and Constraints in Computing Systems*. PhD thesis, Massachusetts Institute of Technology, January 2013.

[79] Henry Hoffmann, Jim Holt, George Kurian, Eric Lau, Martina Maggio, Jason E. Miller, Sabrina M. Neuman, Mahmut Sinangil, Yildiz Sinangil, Anant Agarwal, Anantha P. Chandrakasan, and Srinivas Devadas. Self-aware Computing in the Angstrom Processor. In *Proceedings of the 49th Design Automation Conference*, DAC '12, pages 259–264, New York, NY, USA, 2012. ACM. 10.1145/2228360.2228409.

[80] Anoop Gupta, Andrew Tucker, and Luis Stevens. Making Effective Use of Shared-Memory Multiprocessors: The Process Control Approach. Technical report, Stanford University, 1991.

[81] Janghaeng Lee, Haicheng Wu, Madhumitha Ravichandran, and Nathan Clark. Thread Tailor: Dynamically Weaving Threads Together for Efficient, Adaptive Parallel Applications. In *Proceedings of the 37th International Symposium on Computer Architecture*, ISCA '10, pages 270–279, New York, NY, USA, 2010. ACM. 10.1145/1815961.1815996.

[82] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, and Babak Falsafi. Scale-Out Processors. In *Proceedings of the 39th International Symposium on Computer Architecture*, ISCA '12, pages 500–511, Washington, DC, USA, 2012. IEEE Computer Society. 10.1109/ISCA.2012.6237043.

[83] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-Aware Microarchitecture: Modeling and Implementation. *ACM Trans. Archit. Code Optim.*, 1(1):94–125, March 2004. 10.1145/980152.980157.

[84] Nosayba El-Sayed, Ioan A. Stefanovici, George Amvrosiadis, Andy A. Hwang, and Bianca Schroeder. Temperature Management in Data Centers: Why Some (Might) Like It Hot. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 163–174, New York, NY, USA, 2012. ACM. 10.1145/2254756.2254778.

[85] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The Case for Lifetime Reliability-Aware Microprocessors. In *Proceedings of the 31st International Symposium on Computer Architecture*, ISCA '04, pages 276–287, Washington, DC, USA, 2004. IEEE Computer Society. 10.1109/ISCA.2004.1310781.

[86] Matthew Garrett. Powering Down. *Commun. ACM*, 51(9):42–46, September 2008. 10.1145/1378727.1378740.

[87] Hewlett-Packard Corp., Intel Corp., Microsoft Corp., Phoenix Technologies Ltd., and Toshiba Corp. Advanced Configuration and Power

Interface Specification, Revision 5.0. Retrieved Sept. 1, 2013 from `http://www.acpi.info/spec50.htm`, December 2011.

[88] 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 *Proceedings of the 38th International Symposium on Microarchitecture*, MICRO '39, pages 347–358, Washington, DC, USA, 2006. IEEE Computer Society. 10.1109/MICRO.2006.8.

[89] Venkatesh Pallipadi, Shaohua Li, and Adam Belay. cpuidle – Do nothing, efficiently. . . . In *Proceedings of the Linux Symposium*, pages 119–126, 2007.

[90] Intel Corp. Intel Xeon Processor 3500 Series: Datasheet, Volume 1. Retrieved Sept. 1, 2013 from `http://www.intel.com/content/www/us/en/processors/xeon/xeon-3500-series-vol1-datasheet.html`, July 2009.

[91] Jonathan Corbet. Idle cycle injection. Retrieved Sept. 1, 2013 from `http://lwn.net/Articles/383368/`, April 2010.

[92] Jonathan Corbet. Intel PowerClamp Driver. Retrieved Sept. 1, 2013 from `https://lwn.net/Articles/528124/`, December 2012.

[93] David Brooks and Margaret Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, HPCA '01, pages 171–182, Washington, DC, USA, 2001. IEEE Computer Society. 10.1109/HPCA.2001.903261.

[94] Xiuyi Zhou, Jun Yang, Marek Chrobak, and Youtao Zhang. Performance-Aware Thermal Management via Task Scheduling. *ACM Trans. Archit. Code Optim.*, 7(1), May 2010. 10.1145/1746065.1736070.

[95] Kevin Skadron, Tarek Abdelzaher, and Mircea R. Stan. Control-Theoretic Techniques and Thermal-RC Modeling for Accurate and Localized Dynamic Thermal Management. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture*, HPCA '02, pages

17–26, Washington, DC, USA, 2002. IEEE Computer Society. 10.1109/HPCA.2002.995695.

[96] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[97] Aurelien Jarno. cpuburn. Retrieved Sept. 1, 2013 from `http://packages.debian.org/wheezy/cpuburn`, 2013.

[98] William S. Levine. *The Control Handbook*. CRC Press, 2nd edition, 1996.

[99] Chengmo Yang and Alex Orailoglu. Processor Reliability Enhancement through Compiler-Directed Register File Peak Temperature Reduction. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '09, pages 468–477, Washington, DC, USA, 2009. IEEE Computer Society. 10.1109/DSN.2009.5270305.

[100] Robert Schöne, Daniel Hackenberg, and Daniel Molka. Memory Performance at Reduced CPU Clock Speeds: An Analysis of Current x86-64 Processors. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[101] Michael D. Powell, Mohamed Gomaa, and T. N. Vijaykumar. Heat-and-Run: Leveraging SMT and CMP to Manage Power Density Through the Operating System. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 260–270, New York, NY, USA, 2004. ACM. 10.1145/1024393.1024424.

[102] Leland Chang and Wilfried Haensch. Near-Threshold Operation for Power-Efficient Computing? It Depends. . . . In *Proceedings of the 49th Design Automation Conference*, DAC '12, pages 1159–1163, New York, NY, USA, 2012. ACM. 10.1145/2228360.2228573.

[103] Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. Near-Threshold Voltage (NTV) Design -

Opportunities and Challenges. In *Proceedings of the 49th Design Automation Conference*, DAC '12, pages 1153–1158, New York, NY, USA, 2012. ACM. 10.1145/2228360.2228572.

[104] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation Cores: Reducing the Energy of Mature Computations. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 205–218, New York, NY, USA, 2010. ACM. 10.1145/1736020.1736044.

[105] Seongmoo Heo, Kenneth Barr, and Krste Asanović. Reducing Power Density through Activity Migration. In *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED '03, pages 217–222, New York, NY, USA, 2003. ACM. 10.1145/871506.871561.

[106] Ramkumar Jayaseelan and Tulika Mitra. Dynamic Thermal Management via Architectural Adaptation. In *Proceedings of the 46th Design Automation Conference*, DAC '09, pages 484–489, New York, NY, USA, 2009. ACM. 10.1145/1629911.1630038.

[107] Justin Moore, Jeff Chase, Parthasarathy Ranganathan, and Ratnesh Sharma. Making Scheduling "Cool": Temperature-Aware Workload Placement in Data Centers. In *Proceedings of the Annual Technical Conference*, ATC '05, pages 61–74, Berkeley, CA, USA, 2005. USENIX Association.

[108] Amit Kumar, Li Shang, Li-Shiuan Peh, and Niraj K. Jha. HybDTM: A Coordinated Hardware-Software Approach for Dynamic Thermal Management. In *Proceedings of the 43rd Design Automation Conference*, DAC '06, pages 548–553, New York, NY, USA, 2006. ACM. 10.1145/1146909.1147052.

[109] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-Aware Server Provisioning and Load Dispatch-

ing for Connection-Intensive Internet Services. In *Proceedings of the 5th Symposium on Network Ssytems Design and Implementation*, NSDI'08, pages 337–350, Berkeley, CA, USA, 2008. USENIX Association.

[110] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[111] Erven Rohou and Michael D. Smith. Dynamically Managing Processor Temperature and Power. In *Proceedings of the 2nd Workshop on Feedback-Directed Optimizations*, 1999.

[112] Pratyush Kumar and Lothar Thiele. Cool Shapers: Shaping Real-time Tasks for Improved Thermal Guarantees. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 468–473, New York, NY, USA, 2011. ACM. 10.1145/2024724.2024835.

[113] Nikhil Gupta and Rabi N. Mahapatra. Temperature Aware Energy Management for Real-Time Scheduling. In *Proceedings of the 12th International Symposium on Quality Electronic Design*, ISQED '11, pages 91–96, 2011. 10.1109/ISQED.2011.5770709.

[114] Parthasarathy Ranganathan and Jichuan Chang. Saving the World, One Server at a Time, Together. *IEEE Computer*, 44(5):91–93, May 2011. 10.1109/MC.2011.156.

[115] Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004.

[116] Alberto Scolari, Filippo Sironi, Davide B. Bartolini, Donatella Sciuto, and Marco D. Santambrogio. Coloring the Cloud for Predictable Performance. In *Proceedings of the 4th Symposium on Cloud Computing*, SoCC '13, New York, NY, USA, 2013. ACM. 10.1145/2523616.2525955.

[117] Lamia Youseff, Nathan Beckmann, Harshad Kasture, Charles Gruenwald, David Wentzlaff, and Anant Agarwal. The Case for Elastic Operating System Services in fos. In *Proceedings of the 49th Design Automation*

Conference, DAC '12, pages 265–270, New York, NY, USA, 2012. ACM. 10.1145/2228360.2228410.

[118] Davide B. Bartolini, Filippo Sironi, Martina Maggi, Gianluca C. Durelli, Donatella Sciuto, and Marco D. Santambrogio. Towards a Performance-as-a-Service Cloud. In *Proceedings of the 4th Symposium on Cloud Computing*, SoCC '13, New York, NY, USA, 2013. ACM. 10.1145/2523616.2525933.

[119] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated Control of Multiple Virtualized Resources. In *Proceedings of the 4th European Conference on Computer Systems*, EuroSys '09, pages 13–26, New York, NY, USA, 2009. ACM. 10.1145/1519065.1519068.

[120] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems. In *Proceedings of the 2nd Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. ACM. 10.1145/2038916.2038921.

[121] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3):54–66, May 2008. 10.1109/MM.2008.48.

[122] Reza Azimi, Michael Stumm, and Robert W. Wisniewski. Online Performance Analysis by Statistical Sampling of Microprocessor Performance Counters. In *Proceedings of the 19th International COnference on Supercomputing*, ICS '05, pages 101–110, New York, NY, USA, 2005. ACM. 10.1145/1088149.1088163.

[123] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 121–132, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-406-5. 10.1145/1508244.1508259.

[124] Steven Hofmeyr, Juan A. Colmenares, Costin Iancu, and John Kubiatowicz. Juggle: Proactive Load Balancing on Multicore Computers. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 3–14, New York, NY, USA, 2011. ACM. 10.1145/1996130.1996134.

[125] Heechul Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and Lui Sha. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms. In *Proceedings of the19th Real-Time and Embedded Technology and Applications Symposium*, RTAS '13, pages 55–64, Washington, DC, USA, 2013. IEEE Computer Society. 10.1109/RTAS.2013.6531079.

[126] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism. In *Proceedings of the 13thSymposium on Operating Systems Principles*, SOSP '91, pages 95–109, New York, NY, USA, 1991. ACM. 10.1145/121132.121151.