Exploiting Managed Language Semantics to Optimize for Hardware Heterogeneity

Optimalisatie van heterogene hardware via semantische informatie
in beheerde programmeertalen

Shoaib Akram

Promotoren: prof. dr. ir. L. Eeckhout, dr. J. Sartor
Proefschrift ingediend tot het behalen van de graad van
Doctor in de ingenieurswetenschappen: computerwetenschappen

UNIVERSITEIT
GENT

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. K. De Bosschere
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2018 - 2019

ISBN 978-94-6355-245-5
NUR 980, 987
Wettelijk depot: D/2019/10.500/53

Examination Committee

Prof. Hennie De Schepper, *chair*
Department of Mathematical Analysis,
Faculty of Engineering and Architecture
Ghent University

Prof. Koen De Bosschere, *secretary*
Department of Electronics and Information Systems,
Faculty of Engineering and Architecture
Ghent University

Prof. Lieven Eeckhout, *supervisor*
Department of Electronics and Information Systems,
Faculty of Engineering and Architecture
Ghent University

Prof. Jennifer B. Sartor, *supervisor*
Department of Computer Science,
Vrije Universiteit Brussel

Department of Electronics and Information Systems,
Faculty of Engineering and Architecture
Ghent University

Prof. Luis Ceze
Paul G. Allen School of Computer Science & Engineering,
University of Washington - Seattle

Dr. Kathryn McKinley
Google, USA

Prof. Bart Dhoedt
Department of Information Technology,
Faculty of Engineering and Architecture
Ghent University

Prof. Elisa Gonzalez Boix
Department of Computer Science,
Vrije Universiteit Brussel

Contents

Examination Committee **i**

Acknowledgements **ix**

Samenvatting **xi**

Summary **xv**

List of Figures **xix**

List of Tables **xxiii**

List of Abbreviations **xxv**

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis Contributions 3

1.3 Structure and Overview 7

2 Background **9**

2.1 Hardware 9

2.1.1 Device Scaling Trends 9

2.1.2 Heterogeneous Multicores 10

2.1.3 Hybrid Memories 11

2.2 Software 12

2.2.1 Managed Languages and Runtimes 12

2.2.2 Garbage Collection 13

2.2.3 Generational Garbage Collection. 13

2.2.4 GenImmix 14
2.2.5 Java Virtual Machine 14
2.2.6 Java Performance Evaluation 15

3 GC-Criticality-Aware Scheduling on Heterogeneous Multicores 17
3.1 Introduction 17
3.2 Related Work 19
3.2.1 Scheduling Managed Language Workloads on Heterogeneous Multicores 19
3.2.2 Managed Language Workloads on Multicores 20
3.2.3 Scheduling on Heterogeneous Multicores 20
3.3 Background 21
3.3.1 Concurrent Garbage Collection 21
3.3.2 Garbage Collection on Heterogeneous Multicores 22
3.4 Concurrent GC on Heterogeneous Multicores 23
3.4.1 Generalizing to Different Garbage Collectors 25
3.5 GC-criticality-aware Scheduling 26
3.5.1 Base Schedulers 26
3.5.2 GC-criticality-aware Scheduler 27
3.6 Experimental Setup 30
3.7 Experimental Evaluation 32
3.7.1 Performance 32
3.7.2 Energy Efficiency 36
3.7.3 Scaling Small Core Frequency 36
3.7.4 Larger Core Counts 38
3.7.5 Heap Size Sensitivity Study 38
3.8 GC Criticality in OpenJDK 39
3.9 Summary and Interpretation 40

4 DVFS Performance Prediction with DEP+BURST 43
4.1 Introduction 43
4.2 Related Work 45
4.2.1 DVFS Performance and Power Prediction 45
4.2.2 Scheduling Multithreaded Applications 46
4.2.3 Energy Management 46

4.3 Background and Motivation 46
4.3.1 DVFS Performance Predictors for Sequential Applications . . 47
4.3.2 Challenges in DVFS Performance Prediction for Managed Multithreaded Applications 48
4.3.3 Straightforward Extensions of Prior Work 48
4.4 The DEP+BURST Model 49
4.4.1 Overview 49
4.4.2 Identifying Synchronization Epochs 50
4.4.3 Predicting Performance at a Target Frequency 51
4.4.4 Modeling Store Bursts 52
4.4.5 Implementation Details 53
4.5 Experimental Methodology 53
4.6 Model Evaluation 54
4.6.1 Prediction Accuracy 55
4.6.2 Per-Epoch vs. Across-Epochs CTP 58
4.6.3 Scalability 59
4.7 Case Studies 60
4.7.1 Case Study 1: Energy Minimization under Performance Constraints 60
4.7.2 Case Study 2: Minimizing Full System Energy 68
4.8 Summary and Interpretation 69

5 Kingsguard: Write-Rationing Garbage Collection 71
5.1 Introduction 71
5.2 Related Work 74
5.3 Background 76
5.3.1 Write Barriers 76
5.3.2 Large Objects 77
5.3.3 Object Metadata 77
5.4 Write-Rationing Garbage Collection 77
5.4.1 Kingsguard-nursery for Hybrid Memory 77
5.4.2 Kingsguard-writers for Hybrid Memory 79
5.5 Experimental Methodology 83
5.5.1 Software 83
5.5.2 Hardware and Simulation 85

5.6 Results . . . 87
5.6.1 Simulation Results . . . 88
5.6.2 Real Hardware Results . . . 92
5.7 Summary and Interpretation . . . 98

6 Crystal Gazer: Profile-Driven Write-Rationing Garbage Collection 99
6.1 Introduction . . . 99
6.2 Related Work and Background . . . 101
6.3 Allocation Site as a Write Predictor . . . 102
6.4 Crystal Gazer . . . 103
6.4.1 Overview . . . 104
6.4.2 Profiling . . . 104
6.4.3 Allocation Site Classification . . . 105
6.4.4 Bytecode Generation . . . 107
6.4.5 Heap Organization . . . 108
6.5 Emulation on NUMA Hardware . . . 109
6.5.1 Hardware . . . 110
6.5.2 Heap Layout and Management . . . 110
6.5.3 Space to Socket Mapping . . . 112
6.5.4 Thread to Socket mapping . . . 113
6.6 Experimental Methodology . . . 113
6.7 Results . . . 115
6.7.1 PCM Writes . . . 115
6.7.2 DRAM Capacity . . . 116
6.7.3 Trading Off PCM Writes and DRAM Capacity . . . 117
6.7.4 Allocation Site Analysis . . . 117
6.7.5 Performance . . . 119
6.7.6 Sensitivity Analyses . . . 120
6.7.7 Memory and Demographic Analysis . . . 122
6.7.8 PCM Lifetime and Write Rates . . . 122
6.8 Summary and Interpretation . . . 123

7 Conclusion and Future Work 127
7.1 Conclusion . . . 127

7.2 Future Work 129
7.2.1 Scheduling for Heterogeneous Multicores 129
7.2.2 Performance Prediction 130
7.2.3 Write-Rationing Garbage Collection 130

Bibliography **133**

Acknowledgements

First and foremost, I want to thank my advisor, Lieven Eeckhout, for providing me with an opportunity to be a part of his research group. We met at a time when I was at crossroads in my career. He has since guided me in doing something fruitful with my life. His advice on conducting impactful research, and the extensive feedback on research paper drafts and presentations, has helped me become a better researcher in computer systems. He continually reminds me to aim high and guides me on how to stay afloat in the face of failures. Lieven has been both patient and generous in supporting my desire to pursue an academic career.

Many thanks also to Jennifer Sartor, who served as my Ph.D. co-advisor. Her constant feedback on different elements of the research process helped improve the work presented in this thesis. In the early days of my Ph.D. life, she taught me to persist in solving a challenging problem instead of giving up too early.

During the last few years of my Ph.D., I worked closely with Kathryn McKinley. Kathryn encouraged me to critically reason the role of different layers in the vertical stack in solving challenging computer systems problems. She was instrumental in helping me publish my first top-tier paper. Over the years, I asked for her advice on different facets of research, career planning, teaching, and academic job hunting. Kathryn found time out of her busy schedule to provide me feedback on my writing drafts, presentations, and application material for faculty positions. For all her support, I express my sincere gratitude!

I am grateful to the members of my examination committee, who were supportive during the process of submitting and defending my thesis. I especially thank Luis Ceze and Kathryn McKinley, who flew from Seattle to take part in my internal Ph.D. defense.

I thank all past and present members of the PerfLab at Ghent University for their support, laughs, and critical discussions. During my early days in Ghent, Max and Kristof made my stay very comfortable and shared great laughs (and sometimes pulled off nasty pranks). In retrospect, the arrival of Almutaz Adileh in our research group proved vital to the quality of my research output. His constant feedback on refining research ideas, paper drafts, research talks, and his attention to detail helped me to improve my work continuously.

I am immensely grateful to Wim Heirman for many things. He has supported me during my stay in Ghent, both in research and outside of it. I continued brainstorming

for fresh ideas with him even after he left Ghent University. He supported me to overcome engineering challenges and helped me to finish experiments on time. Legend has it that to this day, three Lynx machines in Kortrijk, enclosed in a rack that nobody knows of, are running simulation jobs that will take forever to finish (meanwhile on a boat in Ghent, two architects are preparing to have another round of St. Bernardus).

I want to thank the staff at Ghent University for their support with administrative tasks. I especially thank Marnix Vermassen for his help with travel logistics. I want to extend thanks to people outside the Ghent University, the staff at the city hall in Ghent, and more generally, the people of Ghent, who made my stay very peaceful, comfortable, and a joy to remember.

I came to Ghent University on the back of two enriching experiences. I would like to remember the friendships I made during my stay in Crete as a junior researcher. I thank Angelos Bilas for allowing me to join FORTH and to be part of the CARV laboratory. He was very generous to spend time with me in discussing new ideas and help with experimental design. I was privileged to spend two years in the Coordinated Science Laboratory at the University of Illinois. My M.S. thesis advisor, Deming Chen, taught me several things, and most importantly, the value of hard work.

Finally, I thank my parents, who always inspired me to achieve big things in life. Their patience and support of the choices I have made in life have been exemplary. My two sisters gave me love, support, inspiration, and happiness that no words can describe. Among other things, they beat me to the title of a Doctor by a margin of several years.

Shoaib Akram
Gent, June 14, 2019

Samenvatting

Moderne hardware wordt in toenemende mate heterogeen als gevolg van recente trends in chiptechnologie. Het traditionele Dennard-schalingsgedrag van transistors is gestopt rond het jaar 2000. Zonder Dennard-schaling leidt een toenemend aantal transistors op de chip tot een vermogenstoename van de volledige chip. Bijgevolg is energie-efficiëntie één van de belangrijkste criteria tijdens het ontwerp en de operatie van microprocessors.

Heterogene meerkernige processors (HMPs) vertonen hoge energie-efficiëntie door processorkernen (Eng. *cores*) met verschillende architecturale karakteristieken te integreren op eenzelfde chip. Typisch worden hoog-performante *out-of-order* processorkernen gecombineerd met energie-efficiënte *in-order* processorkernen. Daarnaast is het mogelijk dynamisch de voedingsspanning en klokfrequentie per processorkern te schalen (Eng. *dynamic voltage and frequency scaling*). Beide vormen van heterogeniteit laten de software toe een taak uit te voeren op de meest efficiënte processorkern.

Conventioneel geheugen (Eng. *dynamic random access memory* of *DRAM*) heeft eveneens te kampen met schalingsbeperkingen. De complexiteit voor het fabriceren van steeds kleinere DRAM-cellen neemt toe waardoor de kostprijs van geheugen sterk is toegenomen. Tegelijkertijd zijn er vandaag heel wat computertoepassingen die steeds grotere volumes aan geheugencapaciteit nodig hebben. Nieuwe niet-volatiele geheugentechnologieën (Eng. *non-volatile memory* of *NVM*) zijn beter schaalbaar dan DRAM en bieden een grotere densiteit en dus een hogere capaciteit, zijn byte-addresseerbaar, vertonen kleine lekstromen en zijn persistent (niet-volatiel). *Phase Change Memory (PCM)* is op dit moment de meest veelbelovende NVM-technologie. De grootste beperkingen van PCM zijn echter de beperkte schrijfbaarheid en de hoge toegangslatentie. Heterogene geheugensystemen combineren DRAM en PCM om op die manier het beste van beide aan te bieden. Mits effectief beheer levert een heterogeen geheugensysteem hoge prestatie, laag vermogenverbruik, hoge densiteit en persistentie.

Met betrekking tot software maken programmeurs steeds vaker gebruik van beheerde programmeertalen zoals Java, C#, Python en Go. Software ontwikkeld in een beheerde programmeertaal biedt overdraagbaarheid aan over platformen door gebruik te maken van een virtuele machine. Dergelijke talen verhogen bovendien de productiviteit van de programmeur via automatisch geheugenbeheer en door de uitvoerbare code dynamisch te genereren en te optimaliseren. Hierdoor beschikt de virtuele machine over heel wat semantische informatie betreffende de computertoepassing.

Deze doctoraatsthesis exploreert de rol die beheerde programmeertalen kunnen spelen in het abstraheren en optimaliseren van heterogene processors en geheugensystemen. Meer specifiek maakt deze doctoraatsthesis vier bijdragen die elk aantonen hoe semantische informatie in beheerde programmeertalen via de virtuele machine geëxploiteerd kan worden in heterogene computersystemen. De thesis levert twee bijdragen rond heterogene processors en twee bijdragen rond heterogene geheugensystemen. Zonder afbreuk te doen aan de algemene toepasbaarheid van de bijdragen, spitst de thesis zich toe op de Java programmeertaal en de bijhorende Java virtuele machine (JVM).

De eerste bijdrage van de thesis betreft het uitvoeren van automatisch geheugenbeheer (Eng. *garbage collection* of *GC*) op heterogene multicore processors (HMPs). GC bevrijdt de programmeur van het manueel beheren van het geheugengebruik en is bijgevolg een kritische component van een beheerde programmeertaal. *Stop-the-world* GC stopt de uitvoering van een computertoepassing om ongebruikt geheugen vrij te geven. Concurrente GC leidt tot betere prestatie door parallel uit te voeren met de toepassing. Echter, indien concurrente GC het geheugen niet snel genoeg vrijgeeft, kan dit toch leiden tot het stopzetten van de toepassing teneinde het geheugen alsnog vrij te geven zodat de toepassing nieuw geheugen kan alloceren. We noemen dit fenomeen GC-kriticiteit.

Deze doctoraatsthesis toont aan dat verschillende Java-computertoepassingen GC-kriticiteit vertonen wanneer concurrente GC uitvoert op een energie-efficiënte in-order processorkern in een HMP. Concurrente GC vertoont echter voldoende parallellisme op instructieniveau en kan dus bijgevolg sneller uitvoeren op een hoog-performante out-of-order processorkern. Op basis van deze observaties stellen we voor concurrente GC dynamisch te beheren. Hierbij wordt concurrente GC uitgevoerd op een energie-efficiënte rekenkern teneinde het energieverbruik te beperken tijdens normale uitvoering. Echter, wanneer GC kritisch dreigt te worden, geeft de virtuele machine een signaal aan het besturingssysteem om GC op een hoog-performante processorkern uit te voeren. Het dynamisch beheren van concurrente GC levert betere prestatie op en verhoogt de energie-efficiëntie.

De tweede bijdrage exploreert prestatieschatting van DVFS voor meerdradige computertoepassingen in beheerde programmeertalen. Prestatieschatting van DVFS is een techniek die toelaat de meest energie-efficiënte voedingsspanning en klokfrequentie te bepalen voor een gegeven toepassing. Bestaande methodes zijn enkel nauwkeurig voor sequentiële computertoepassingen geschreven in niet-beheerde programmeertalen zoals C en C++. De reden voor de onnauwkeurigheid voor meerdradige toepassingen in beheerde programmeertalen is tweeledig: (i) synchronisatie in meerdradige toepassingen leidt tot afhankelijkheden tussen draden, en (ii) computertoepassingen in beheerde programmeertalen voeren vaak een opeenvolging van schrijfoperaties uit als gevolg van het automatisch geheugenbeheer.

In deze thesis stellen we DEP+BURST voor, een nieuwe DVFS-prestatieschatter die de impact van afhankelijkheden tussen draden en de opeenvolging van schrijfoperaties nauwkeurig voorspelt onder DVFS. DEP+BURST onderschept synchronisatie tussen draden in een meerdradige toepassing en gebruikt vervolgens een analytisch model om de impact van DVFS op synchronisatie te schatten. Daarnaast modelleert DEP+BURST de mogelijke invloed van een opeenvolging van schrijfoperaties op de

uitvoeringstijd van het computerprogramma. We gebruiken DEP+BURST om het energieverbruik dynamisch te beheren en te reduceren.

De derde bijdrage in de thesis toont aan hoe automatisch geheugenbeheer aangewend kan worden om de levensduur van PCM-geheugen te verbeteren in een heterogeen geheugensysteem. Bestaande technieken op het niveau van de hardware en het besturingssysteem zijn inefficiënt omdat ze opereren op een grote granulariteit en/of reactief zijn. Deze thesis stelt schrijfrantsoenerend geheugenbeheer (Eng. *write-rationing garbage collection*) voor. Objecten worden gealloceerd in een heterogeen geheugensysteem volgens hun schrijfintensiteit: objecten die vaak geschreven worden, worden gealloceerd in DRAM teneinde de levensduur van PCM te verbeteren; objecten die vaak gelezen (en weinig of niet geschreven) worden, worden gealloceerd in PCM teneinde gebruik te maken van PCM's hoge densiteit.

Empirisch onderzoek in deze thesis rond generationele GC toont aan dat objecten die gealloceerd worden in de zogenaamde *nursery* geheugenruimte alsook een beperkt aantal objecten in de *mature* geheugenruimte meest frequent geschreven worden. Op basis van deze observaties stellen we twee implementaties van schrijfrantsoenerend geheugenbeheer voor. Kingsguard-nursery (KG-N) plaatst de nursery in DRAM en de rest van de geheugenruimte in PCM. KG-N reduceert het aantal schrijfoperaties naar PCM drastisch t.o.v. een geheugensysteem bestaande uit enkel PCM. Kingsguard-writers (KG-W) gaat een stap verder en plaatst naast de nursery ook nog een observatieruimte in DRAM. Objecten niet geschreven worden in de observatieruimte worden vervolgens gealloceerd in PCM. KG-W reduceert het aantal schrijfoperaties t.o.v. KG-N maar introduceert een kleine impact op prestatie. We tonen aan dat schrijfrantsoenerend geheugenbeheer de levensduur van PCM drastisch verlengt en bijgevolg een veelbelovende techniek is voor het beheer van heterogene geheugensystemen.

De vierde bijdrage van de thesis verfijnt schrijfrantsoenerend geheugenbeheer op basis van de observatie dat het schrijfgedrag van objecten nauwkeurig te voorspellen is op basis van de plaats in de code waar de objecten gealloceerd worden. We stellen Crystal Gazer voor die op basis van profilering van de allocatiesites objecten classificeert als al dan niet schrijfintensief. Moderne mobiele toepassingen alsook servertoepassingen voeren heel vaak steeds dezelfde code uit waardoor profilering te verantwoorden is. Via profilering en statische voorspelling elimineert Crystal Gazer de overhead van KG-W voor het monitoren van het schrijfgedrag van objecten.

Crystal Gazer gebruikt de profileringsinformatie om schrijfintensieve objecten proactief te alloceren in DRAM en de rest van de objecten in PCM. Het resultaat is geheugenbeheer met een verwaarloosbare overhead en een significante reductie in het aantal schrijfoperaties naar PCM i.v.m. KG-W. Crystal Gazer laat toe verschillende heuristieken te gebruiken om objecten al dan niet als schrijfintensief te classificeren. Deze heuristieken laten toe een Pareto-optimale afweging te maken tussen de levensduur van PCM en de benodigde DRAM-capaciteit.

De vier bijdragen in deze thesis tonen collectief aan dat het gebruik van semantische informatie in een virtuele machine voor beheerde programmeertalen effectief is voor het optimaliseren van heterogene processors en geheugensystemen.

Summary

Modern hardware is becoming increasingly heterogeneous. Hardware heterogeneity is a response to recent semiconductor device scaling trends. The traditional shrinking of transistor sizes under Dennard's rule stopped in the early 2000s. Without Dennard's scaling, as transistor budgets increase, so does the power they collectively consume. As a result, energy efficiency has become a first-order concern during the design and operation of processors.

Heterogeneous multicore (HM) processors promise energy efficiency by combining cores with different architectural capabilities on the same chip. In production HMs, big (high performance) cores execute instructions out-of-order, whereas small (low-power) cores execute instructions in program order. Even cores with similar architectural capabilities have dynamic voltage and frequency scaling (DVFS) which allows for simultaneously changing a core's voltage and frequency to save energy. Altogether, HMs expose a choice to software to execute each task on the most suitable core.

On the memory side, dynamic random access memory (DRAM) is also facing scaling limitations. The complexity of manufacturing ever-smaller DRAM cells is increasing. As a consequence, main memory cost is now a serious concern. At the same time, the capacity demands of modern applications are continually increasing. Emerging non-volatile memory (NVM) technologies are more scalable than DRAM and offer higher capacity, byte-addressability, low leakage power, and persistence. Phase change memory (PCM) is currently the most promising NVM technology. The main disadvantages of PCM are its limited write endurance and high latency. Heterogeneous (hybrid) memory combines DRAM and PCM to offer the best of both technologies. With proper management, a hybrid memory system provides high performance, low power, high density, and persistent memory.

On the software side, programmers exceedingly prefer managed languages (e.g., Java, C#, Python, Go) for software development. Managed languages offer portability by executing on top of a virtual machine. These languages also provide various services to aid programmer productivity in the form of a managed runtime. Managed runtimes abstract hardware complexity and interact closely with the application. Their close interaction with the application means they contain rich semantic information about application needs and behaviors.

This thesis explores the role of managed runtimes in abstracting the complexity of heterogeneous processors and memories. More specifically, this thesis makes four distinct contributions that show how semantic information in the managed runtime

helps to exploit heterogeneous processors and memories better. We present two contributions for managing HMs and two contributions for managing hybrid memories. Without loss of generality, the managed runtime environment we consider in this thesis is the Java Virtual Machine (JVM) which is the virtual machine implementation for the popular Java programming language.

The first contribution in this thesis explores the scheduling of garbage collection on HMs with big and small cores to improve energy efficiency. Garbage collection (GC) relieves the programmer from the burden of manually freeing memory. GC is a critical service that managed languages offer to programmers. Stop-the-world collectors stop the application to free unused memory on the heap but incurs a performance penalty. Concurrent GCs run concurrently with the application and have gained popularity with the arrival of multicore processors. They improve performance because they do not require the application to stop. However, if the concurrent collector cannot free memory fast enough to keep up with application allocation, it could stop the application. We call this GC-criticality.

This thesis shows that several Java applications exhibit GC-criticality when concurrent GC runs on the small cores of an HM. Fortunately, GC exhibits instruction-level parallelism and can run faster on a big out-of-order core. Based on these observations, we propose GC-criticality-aware scheduling. In our proposal, the concurrent GC executes on small cores to conserve energy during normal execution. However, when concurrent GC becomes critical, the JVM delivers a criticality signal to the OS scheduler. The OS scheduler, in turn, increases the priority of concurrent GC for big cores. GC-criticality-aware scheduling improves both the performance and the energy efficiency of Java applications running on HMs compared to state-of-the-art schedulers for managed applications.

The second contribution in this thesis explores DVFS performance prediction for managed multithreaded applications. DVFS predictors guide system software in choosing the most appropriate DVFS setting for a running application or a specific phase of it. This thesis shows that prior DVFS predictors are only accurate for single-threaded native applications written in C and C++. The reasons for their inaccuracy are two-fold: (1) synchronization in multithreaded applications leads to inter-thread dependences, and (2) managed applications execute bursts of store operations due to memory management.

Due to inter-thread dependences, changing the DVFS setting of one core (thread) impacts the execution of dependent threads. In addition, a burst of store operations stalls a core which affects DVFS. This thesis proposes DEP+BURST, a new predictor, which takes into account both inter-thread dependences and store bursts. DEP+BURST intercepts synchronization activity in a multithreaded application. It then uses a simple analytical model to predict the performance of an application at a different DVFS setting. Furthermore, DEP+BURST accurately models the impact of store bursts on DVFS. We integrate DEP+BURST in an energy manager to demonstrate energy savings using DVFS for popular Java applications.

The third contribution in this thesis explores the role of managed runtimes, and garbage collection in particular, in mitigating PCM wear-out to improve its lifetime in hybrid memories. Existing hardware and OS approaches to mitigate PCM wear-out

suffer from two drawbacks: (1) they operate at coarse-grain page granularity, and (2) they are reactive. This thesis proposes write-rationing garbage collection that reorganizes fine-grained objects in hybrid memory with the aim to keep frequently written objects in DRAM to mitigate PCM wear-out. They keep read-mostly objects in PCM to exploit its capacity.

Our empirical analysis of popular Java applications using a generational garbage collector shows that nursery objects and a small number of mature objects get most of the writes. Based on these observations, we propose two write-rationing Kingsguard collectors. Kingsguard-nursery (KG-N) places the nursery in DRAM and survivors in PCM, significantly reducing PCM writes over a PCM-only memory system. Kingsguard-writers (KG-W) places the nursery in DRAM and survivors in a DRAM observer space. It dynamically monitors all writes to nursery survivors and moves unwritten objects to PCM. KG-W further reduces PCM writes but incurs some performance overhead compared to KG-N. Our results show that write-rationing garbage collection improves PCM lifetime and is a promising approach to managing hybrid memories.

The fourth contribution in this thesis contributes further to write-rationing garbage collection for hybrid memories. Our analysis shows that mature-object writes in Java applications are predictable on a per allocation-site basis. Based on this observation, we propose Crystal Gazer, which uses static profiling of allocation sites to identify frequently written objects. Modern mobile and server applications repeatedly execute, which makes profiling realistic. Crystal Gazer overcomes the main disadvantage of KG-W. Specifically, the cost to dynamically monitor mature-object writes in KG-W is an overhead in performance.

Crystal Gazer uses the profiling information to proactively allocate highly written objects in DRAM and the rest in PCM. The result is a garbage collector with negligible overhead and a more significant reduction in PCM writes compared to KG-W. Crystal Gazer uses different heuristics to classify allocation sites as DRAM or PCM. These classification heuristics open up Pareto-optimal trade-offs between PCM lifetime and DRAM capacity.

The outcomes of the four contributions in this thesis show that semantic information in the managed runtime can help to exploit heterogeneous processors and memories better. The work presented in this thesis necessitates no changes to the programming languages or models, requires minimal OS support, and needs no extra hardware support. These advantages give developers both the productivity of managed languages and automated ways to utilize HMs and hybrid memories efficiently.

List of Figures

1.1 Layers in a managed runtime environment running on top of heterogeneous hardware. 2

2.1 An example heterogeneous multicore architecture from ARM. 10
2.2 Our assumed memory and storage hierarchy. 12

3.1 Threads and phases of application execution using Jikes' concurrent collector. 22
3.2 Execution time increase when concurrent GC threads are run on small versus big cores. 23
3.3 CPI stacks of the application and concurrent garbage collector threads. 24
3.4 Execution time increase when GC threads are run on small versus big cores. 26
3.5 Picture depicting two schedulers across heterogeneous architectures using four application and two GC threads. 27
3.6 A single sampling interval in our GC-criticality-aware scheduler. . . . 29
3.7 Execution time reduction of adaptive and *gc-fair* schedulers compared to *gc-on-small*. 33
3.8 Stop-the-world phase behavior over time for pmd4 on 1B3S with our adaptive scheduler. 35
3.9 Stop-the-world phase behavior over time for antlr on 1B3S with our adaptive scheduler. 35
3.10 Execution time reduction across benchmarks per category for different T_s and I_{max} values. 36
3.11 Percentage reduction in energy-delay product with the adaptive scheduler. 37
3.12 Percentage execution time reduction for 3B1S when scaling down the frequency of the small core. 37
3.13 Execution time reduction with our adaptive scheduler for various heterogeneous architectures with six total cores. 38

3.14 Execution time reduction with our adaptive scheduler with varying heap sizes. 39
3.15 Total execution time increase for OpenJDK when the GC threads, isolated on a separate socket, are run at 1.6 GHz versus 3.0 GHz. . . . 40

4.1 Showing the analytical modeling of a multithreaded application phase using DEP. 49
4.2 Per-benchmark prediction at higher frequency from a 1 GHz baseline. 56
4.3 Per-benchmark prediction at lower frequency from a 4 GHz baseline. . 57
4.4 Comparing per-epoch versus across-epoch critical thread prediction. . 59
4.5 The accuracy of DEP+BURST for different thread counts. 60
4.6 Example illustrating the energy manager using DVFS performance prediction. 61
4.7 Reduction in energy consumption using DEP+BURST. 62
4.8 Frequency settings chosen by the energy manager for xalan and sunflow for a slowdown threshold of 5%. 63
4.9 Reduction in energy consumption achieved by our energy manager compared to the static optimal (Static-Opt) for a slowdown threshold of 10%. 64
4.10 Per-benchmark slowdown for different frequency step settings with and without modeling excess-time. 65
4.11 Per-benchmark reduction in energy consumption for different frequency step settings with and without modeling excess-time. 66
4.12 Per-benchmark slowdown for different values of hold-off. 67
4.13 Per-benchmark slowdown for different quantum lengths. 67
4.14 Per-benchmark reduction in energy consumption when the energy manager optimizes for total system energy. 69

5.1 Lifetime in years for a 32 GB PCM-only system. 73
5.2 Distribution of writes to nursery and mature objects in Java applications. 73
5.3 The main memory heap organizations of Kingsguard write-rationing collectors. 78
5.5 Increase in PCM lifetime with Kingsguard collectors. 88
5.6 Reduction in PCM writes using Kingsguard collectors. 89
5.7 Comparison of Kingsguard collectors to OS-managed write partitioning (WP). 89
5.8 Comparing the energy-delay product of Kingsguard to a DRAM-only and PCM-only system. 90

5.9 Breaking down performance overheads in Kingsguard collectors. . . . 90
5.10 Understanding the sources of writes to PCM. 92
5.11 The writes to PCM measured in an architecture-independent manner. . 93
5.12 Performance overhead with Kingsguard collectors. 94
5.13 DRAM consumption over time with KG-W. 97

6.1 The write homogeneity of allocation sites. 102
6.2 Cumulative distribution of mature writes and heap volume by allocation site for Pjbb and Page Rank. 103
6.3 Overview of Crystal Gazer. 104
6.4 Example of a write-intensity trace. 106
6.5 Heap organizations in Crystal Gazer. 108
6.7 Organization of our proposed Java heap in hybrid DRAM-PCM memory. 110
6.8 Our NUMA platform for emulating hybrid memory. 111
6.9 PCM writes with Crystal Gazer normalized to KG-N. 116
6.10 DRAM space usage with Crystal Gazer and Kingsguard-writers. . . . 116
6.11 Pareto-optimal configurations for CGZ-S compared to KG-W in terms of PCM writes versus DRAM capacity. 117
6.12 Characterization of DRAM-labeled allocation sites. 118
6.13 Understanding the reduction in PCM writes on a per allocation site basis for Pjbb. 118
6.14 Execution time with Crystal Gazer normalized to KG-N. 119
6.15 Showing the impact on PCM writes and DRAM capacity from changing the two Crystal Gazer heuristics to classify allocation sites. 120
6.16 PCM write rates in MB/s for all of our benchmarks using various write-rationing garbage collectors. 123

List of Tables

3.1 GC boost states supported by the GC-criticality-aware scheduler. . . . 29

3.2 Simulated system parameters to evaluate the GC-criticality-aware scheduler. 30

3.3 Benchmark and heap sizes to evaluate the GC-criticality-aware scheduler. 31

4.1 Benchmarks to evaluate DEP+BURST. 54

4.2 Simulated system parameters to evaluate DEP+BURST. 55

5.1 The evaluated Kingsguard collector configurations. 83

5.2 Simulated system parameters to evaluate the Kingsguard collectors. . 84

5.3 Write rates (estimated) in GB/s and their scaling behavior on a real hardware platform. 87

5.4 Object demographics and heap statistics with Kingsguard collectors. . 96

6.1 Spaces in the Kingsguard and the Crystal Gazer collectors and space to socket mapping on NUMA hardware. 112

6.2 Object demographics and heap statistics with Crystal Gazer. 121

List of Abbreviations

GC Garbage Collector

CMS Concurrent Mark Sweep

CPI Cycles Per Instruction

DRAM Dynamic Random Access Memory

EDP Energy-Delay Product

FCFS First Come First Served

ILP Instruction-Level Parallelism

MLP Memory-Level Parallelism

KB Kilo Byte

MB Mega Byte

GB Giga Byte

LRU Least Recently Used

RVM Research Virtual Machine

STW Stop The World

DVFS Dynamic Voltage and Frequency Scaling

ISA Instruction Set Architecture

ROB Re Order Buffer

ITRS International Technology Roadmap for Semiconductors

JVM Java Virtual Machine

NVM Non-Volatile Memory

PCM Phase Change Memory

OS Operating System

WP Write Partition

LLC Last-Level Cache

NUMA Non-Uniform Memory Access

GHz Giga Hertz

LOS Large Object Space

KG Kingsguard

CGZ Crystal Gazer

LOO Large Object Optimization

MDO Meta Data Optimization

Chapter 1

Introduction

1.1 Motivation

Semiconductor device scaling trends have ushered modern hardware into an era of heterogeneity. Heterogeneous computing combines nodes with different capabilities in the same system to promise performance and energy efficiency. More recently, heterogeneous memory systems have emerged that combine different memory technologies to deliver scalable main memory systems.

Two device scaling trends motivate the ongoing shift to hardware heterogeneity. First, the combination of Moore's law and Dennard scaling that delivered greater transistor budgets within reasonable power envelopes has come to an end. As a result, energy efficiency is a critical concern during the design and operation of electronic devices. In recent times, Moore's law has slowed, raising the cost of manufacturing, and leading scientists on the look for more scalable device technologies.

Manufacturers have introduced heterogeneous multicore (HM) processors to manage a processor's power. HMs combine cores with different capabilities to offer power versus performance tradeoffs. Production HMs consist of two core types. Big cores execute instructions out of the normal program order to deliver high performance. Small cores execute instruction in-order and consume less power than big cores. Even cores with similar capabilities have dynamic voltage and frequency scaling (DVFS). DVFS allows to simultaneously change a processor's voltage and frequency to save energy or boost performance.

Heterogeneity is not limited to processors alone. Our memory systems are also becoming heterogeneous. The main reason is the scalability challenge facing dynamic random access memory (DRAM). DRAM has served as the primary memory technology of the last several decades. In recent years, scaling DRAM cells to smaller dimensions is becoming complicated. The main memory scaling challenge is worrisome because modern applications have an insatiable need for memory capacity. Scientists have been exploring emerging non-volatile memory (NVM) technologies to replace DRAM. The most promising NVM technology is phase change memory

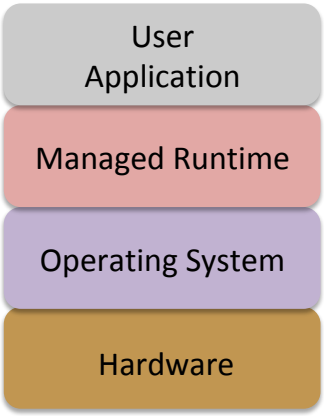


Figure 1.1: Layers in a managed runtime environment running on top of heterogeneous hardware.

(PCM). PCM is byte-addressable and scales better than DRAM. Unfortunately, PCM has two main drawbacks: (1) access latency is high, and (2) write endurance is low. Hybrid memory combines DRAM and PCM to offer main memory systems with high performance, endurance, capacity, and energy efficiency.

Heterogeneous processors and memories expose a variety of tradeoffs to software. Software must take advantage of the emerging hardware heterogeneity. This thesis explores the role of software in utilizing emerging heterogeneous processors and memories efficiently.

On the software side, programmers increasingly prefer managed languages such as Java, C#, Python, Visual Basic, and JavaScript. Managed applications execute on a virtual machine to abstract hardware details and provide platform independence. Managed applications also benefit from a range of services offered by a managed runtime. These services include garbage collection, just-in-time-compilation, memory safety, and network mobility. Garbage collection (GC) is one of the most critical services in a managed language. GC relieves the programmer from the burden of manually freeing heap memory. The result is fast development due to fewer memory related bugs. A famous managed runtime is the virtual machine implementation of the Java programming language, namely the Java Virtual Machine (JVM).

Figure 1.1 shows the different abstraction layers in a managed runtime environment. Managed runtimes interact closely with the user application and abstract hardware and operating system details. Thus, these runtime environments contain rich semantic information about application behaviors. This semantic information can potentially help us to utilize hardware heterogeneity better. Thus, managed applications running on top of heterogeneous hardware offer an opportunity. This thesis answers the question,"how can we exploit the semantic information in a managed language runtime to improve the utility of heterogeneous processors and memories."

This thesis presents four contributions that use semantic information to utilize heterogeneous hardware better. In all contributions, the programming model is

left unchanged. The managed runtime communicates information to the underlying operating system, which in turns forms better policies to manage hardware. We demonstrate our proposed techniques using the Java programming language. However, these techniques are generalizable to other managed languages and runtimes. The next section discusses the key contributions of this thesis.

1.2 Thesis Contributions

Contribution #1: Scheduling Concurrent Garbage Collection on Heterogeneous Multicores.

In a managed runtime, the CPU resource demands of application and service threads change over time. Our particular focus here is the garbage collection service. Garbage collection threads benefit from a more powerful CPU during phases when the application rapidly allocates memory on the heap. Similarly, the application threads go through compute-intensive and memory-intensive phases. Compute-intensive phases benefit from a more powerful CPU than memory-intensive phases.

Modern garbage collectors run concurrently with the application. This better exploits multicore processors that have more than one CPU core on a chip. In a managed environment running on top of a heterogeneous multicore with big and small cores, there are co-executing application and concurrent garbage collection threads.

Prior literature proposes to run garbage collection (GC) on small cores due to two reasons: (1) GC is not on the application's critical path, and (2) GC does not benefit from big cores in a heterogeneous multicore. We find this scheduling policy non-optimal for popular Java applications. Specifically, our work shows that GC can become critical to an application's performance, and especially during phases when the application rapidly allocates memory. If left to run on the small cores, GC lags in freeing up heap memory, and the application eventually stops. This increases GC overhead and degrades overall application performance. Contrary to intuition, our work also shows that garbage collection exposes instruction-level parallelism that the highest-performing cores in an heterogeneous multicore can exploit.

The first contribution of this thesis contributes a new scheduler for managed applications running on top of heterogeneous multicores. The proposed scheduler communicates when GC becomes critical to the OS via a GC-criticality signal. In turn, the OS increases the priority of GC threads to run on the big core. Our GC-criticality-aware scheduler results in better performance and energy efficiency on a range of heterogeneous multicore processors for Java applications. This is one of the first works to exploit semantic information in language runtimes to guide the OS in making the best scheduling decisions for heterogeneous multicores. This work appeared in the ACM Transactions on Architecture and Code Optimization in 2016.

> S. Akram, J. B. Sartor, K. V. Craeynest, W. Heirman, and L. Eeckhout, "Boosting the Priority of Garbage: Scheduling Collection on Heterogeneous Multicore Processors," ACM Transactions on Architecture and Code Optimization (TACO), vol. 13, no. 1, 2016.

We also compare GC-criticality-aware scheduling to prior work on fairness-aware scheduling that proposes equally distributing big core cycles among all threads on a heterogeneous multicore. I contributed to the implementation and evaluation of fairness-aware scheduling which appeared in the International Conference on Parallel Architectures and Compilation Techniques in 2013.

> K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout,"Fairness-Aware Scheduling on Single-ISA Heterogeneous Multi-Cores," International Conference on Parallel Architectures and Compilation Techniques (PACT), 2013.

The benefits of GC-criticality-aware scheduling show that the managed runtime can guide the system software in making better scheduling decisions for heterogeneous multicores.

Contribution #2: DVFS Performance Prediction for Managed Multithreaded Applications.

Dynamic voltage and frequency scaling (DVFS) is another knob to trade power for performance on a heterogeneous multicore processor. To exploit DVFS requires an accurate DVFS performance predictor. Our analysis of the state-of-the-art performance predictors shows that they are not accurate for multithreaded managed applications.

We found two reasons for this inaccuracy. First, managed runtimes are inherently multithreaded. This is because the application and services (such as garbage collection) execute as separate contexts. On top of that, today's applications are multithreaded to best utilize the multiple CPUs (cores) on a multicore processor. All these threads communicate with each other via shared data. The code that operates on shared data, a.k.a., critical section, is protected by locks for correct execution. This leads to inter-thread dependences in a multithreaded managed environment. Therefore, changing the speed of one thread using DVFS impacts the execution of dependent threads. The result is an overall impact on performance that is complicated to model.

In addition to inter-thread dependences, the second reason for inaccuracy is that managed applications issue bursts of store operations. This happens during zero initialization of memory that ensures memory safety in managed languages. Another reason is garbage collection activities that move objects around. Existing predictors ignore store operations assuming they are not on the critical path.

The second contribution of this thesis is an accurate DVFS performance predictor for multithreaded managed applications. Our predictor, namely *DEP+BURST*, continuously intercepts the synchronization activity in a multithreaded application. It then uses analytical modeling to reconstruct the execution of the application at a different frequency. This allows DEP+BURST to report the predicted execution time of the application at a different frequency. DEP+BURST also correctly models the performance impact of store operations.

We use DEP+BURST to propose two energy managers that aim to optimize the energy consumed by an application under a user-specified (maximum) slowdown threshold. The DEP+BURST model, along with an energy manager that optimizes the energy consumed by the processor alone appeared in the IEEE International

Symposium on Performance Analysis of Systems and Software in 2016. This paper was chosen as one of the three candidates for the award of best paper at the conference.

> S. Akram, J. B. Sartor, and L. Eeckhout, "DVFS performance prediction for managed multithreaded applications," IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2016.

An extended version of the above paper appeared in the IEEE Transactions on Computers in 2017. The extended version discusses the scalability of DEP+BURST with increasing thread counts; a new energy manager that considers memory system energy in addition to processor energy; and a wide range of sensitivity studies with the energy managers.

> S. Akram, J. B. Sartor, and L. Eeckhout, "DEP+BURST: Online DVFS Performance Prediction for Energy-Efficient Managed Language Execution," IEEE Transactions on Computers, vol. 66, no. 4, Apr. 2017.

This work shows that existing DVFS predictors can not accurately predict the performance impact of DVFS for multithreaded managed environments. Using semantic information from the runtime environment allows DEP+BURST to accurately predict the performance impact of DVFS for popular Java applications and help save energy.

Contribution #3: Write-Rationing Garbage Collection for Hybrid Memories.

The working set sizes of applications are continually increasing. Unfortunately, DRAM scaling has slowed down in recent years. The resulting manufacturing complexity is increasing the cost of main memory. Furthermore, DRAM based memory systems consume significant power. To fulfill main memory demands at a reasonable cost, scientists are exploring alternatives. Emerging non-volatile memory (NVM) is persistent, byte-addressable, scalable, and consumes less power. It has two main disadvantages: (1) latency is higher than DRAM, (2) write endurance is low. Endurance is a hard problem because each write changes the material form of NVM cells causing them to wear out. Unfortunately, wear-out limits overall memory lifetime. Hybrid memories combine DRAM and NVM to deliver the best of both technologies.

The third contribution of this thesis is write-rationing garbage collection for hybrid memories. Write-rationing garbage collection uses managed language semantics to guide allocation of program (heap) memory in DRAM and NVM with the aim of improving NVM lifetime. Our analysis of 15 popular Java applications shows that newly allocated objects and a small number of old objects are frequently written. This thesis proposes two write-rationing (Kingsguard) garbage collectors, namely Kingsguard-nursery and Kingsguard-writers. They both place the nursery for newly allocated objects in DRAM. In addition, Kingsguard-writers (KG-W) dynamically monitors writes to nursery survivors in a new *observer* space. On an observer collection, KG-W copies a small number of frequently written objects to DRAM, and copies the remaining objects to NVM.

KG-W performs two additional optimizations to protect NVM from writes. It places object meta-data in a new space in DRAM. The managed runtime writes to

object meta-data during a full-heap garbage collection. In addition, KG-W performs an optimization targeted at large objects. State-of-the-art collectors allocate large objects in a special region of the mature space. KG-W allocates some large objects first in the nursery to give them time to die. It then copies them to special regions for large objects in DRAM or NVM. The two optimizations, in addition to dynamically monitoring objects, protect NVM from writes. Our results indicate orders of magnitude improvements in NVM lifetimes using KG-W.

Prior hardware and OS approaches to mitigate NVM wear-out have the disadvantage that they work with *coarse-grained* pages several kilobytes in size. Our analysis shows that KG-W leads to better NVM lifetimes (for similar or less DRAM usage) compared to prior approaches.

Write-rationing garbage collection appeared in the ACM SIGPLAN Conference on Programming Language Design and Implementation in 2018. This work won the NVMW Memorable Paper Award at the Annual Non-Volatile Memories Workshop in 2019.

> S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout, "Write-Rationing Garbage Collection for Hybrid Memories," ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2018.

This work shows that managing hybird memories using semantic information in the managed language runtime is promising and can make NVM practical as main memory. Our proposed Kingsguard collectors requires minimal OS support and no changes to the programming model.

Contribution #4: Profile-Driven Write-Rationing Garbage Collection for Hybrid Memories.

The fourth and final contribution of this thesis is a profile-driven variant of write-rationing garbage collection for hybrid memories. Profile-driven write-rationing garbage collection uses offline profiling of objects' write behavior to place highly written objects in DRAM. The profile-driven collector proposed in this thesis is called Crystal Gazer. Mobile and server applications execute repeatedly which motivates Crystal Gazer and profiling to manage hybrid memories.

Crystal Gazer overcomes the three drawbacks of the previously described Kingsguard-writers garbage collector. Specifically, KG-W dynamically monitors object writes to find frequently written objects. Monitoring is highly effective but sometimes hurts performance. KG-W is also reactive and relies on the accurate discovery of highly written objects in a limited time window. Finally, KG-W consumes excessive DRAM capacity.

This thesis shows that writes in popular Java applications are predictable on a per allocation-site basis. A few allocation sites allocate the majority of highly written objects. We first profile object writes and their allocation sites. We use two heuristics to classify allocation sites as read-mostly (NVM) or highly written (DRAM). Crystal Gazer uses the allocation-site advice at runtime to place a small number of highly written objects in DRAM. It places the majority of read-mostly objects in NVM exploiting the capacity advantage of NVM. Unprofiled allocation sites are labeled

as NVM by default. Leveraging offline profiling overcomes the main disadvantages of KG-W. Crystal Gazer eliminates more NVM writes than KG-W while consuming similar or less DRAM capacity.

A key feature of Crystal Gazer is its ability to tradeoff NVM writes (lifetime) for DRAM capacity by using different heuristics to classify allocation sites, while requiring only one profiling run. Our experimental analysis shows that Crystal Gazer provides Pareto-optimal tradeoffs whereas KG-W provides a single sub-optimal operating point. Crystal Gazer appeared in the International Conference on Measurement and Modeling of Computer Systems in 2018.

> S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout, "Crystal Gazer: A Profile-Driven Garbage Collector to Manage Hybrid Memories," ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), 2019.

A preliminary workshop paper analyzed the prediction accuracy of different object attributes, i.e., type, size, and allocation site, for finding frequently written objects.

> S. Akram, K. S. McKinley, J. B. Sartor, and L. Eeckhout, "Managing Hybrid Memories by Predicting Object Write Intensity," International Conference on the Art, Science, and Engineering of Programming (<Programming18> Companion), 2018.

We propose and use an emulation platform built using real hardware to evaluate Crystal Gazer. The details of the emulation platform appeared in the IEEE International Symposium on Performance Analysis of Systems and Software in 2019.

> S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout, "Emulating and Evaluating Hybrid Memory for Managed Languages on NUMA Hardware," IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2019.

This work shows that offline profiling to discover access patterns of objects in managed applications is practical. Furthermore, the profiling advice can guide garbage collectors in exploiting hybrid memory. When optimizing for memory lifetimes, offline profiling can help save significantly more writes to PCM than the previously proposed Kingsguard collectors.

1.3 Structure and Overview

The next chapter discusses the background on contemporary software and hardware trends related to this thesis, heterogeneous processors and memories, and managed runtime environments.

The remaining thesis is divided into two main parts. The first part is devoted to optimizing the performance and efficiency of managed workloads on heterogeneous

processors. Chapter 3 discusses GC-criticality-aware scheduling that exploits semantic information in the Java runtime to schedule application and concurrent collector threads on a heterogeneous multicore. Chapter 4 describes DEP+BURST, a new DVFS performance predictor for multithreaded managed applications. In the same chapter, we discuss two applications of DEP+BURST that aim to save the energy consumed by managed language applications.

The second part of the thesis is devoted to the management of hybrid memories consisting of DRAM and non-volatile memory (NVM). Chapter 5 and Chapter 6 introduce two write-rationing garbage collectors for hybrid memories. These collectors aim to improve NVM lifetime by guiding a majority of the writes in managed applications to DRAM. Chapter 5 uses dynamic monitoring to protect NVM from writes. Unfortunately, dynamic monitoring increases the execution time of managed applications. Chapter 6 proposes to use offline profiling of allocation sites to place highly written objects in DRAM.

Chapter 7 concludes this thesis with a summary of the main contributions and directions for future work.

Chapter 2

Background

This chapter discusses the essential background in hardware and software. We discuss semiconductor device scaling trends that encourage hardware heterogeneity. We then introduce heterogeneous multicore (HM) processors and hybrid memory systems. We present the reasons for the popularity of managed languages and discuss garbage collection, a key feature of managed languages. We end this section with a discussion of the managed runtime environment we use in the rest of this thesis.

2.1 Hardware

2.1.1 Device Scaling Trends

In 1965, Gordon Moore predicted that transistor counts on chips would double every two years [93] (known as Moore's law). Moore's prediction held for several decades and enabled the successive processor generations to offer more computing power. However, chips would dissipate much heat if the power density of the shrinking transistors would remain the same. In 1974, Robert H. Dennard observed that as transistors were scaled down, their power density remained constant, as both voltage and current scaled down with length [37] (known as Dennard scaling). The combination of Moore's law and Dennard scaling delivered better performing processors every few years within reasonable power budgets. Power-efficiency was a secondary concern.

The continuing increase in manufacturing complexity has slowed down Moore's law in recent times. On the other hand, Dennard scaling has completely stopped. With Dennard scaling, the transistor dimensions and the supply voltage is scaled down by the same factor. The threshold voltage is also scaled down to increase switching speed. However, scaling down threshold voltage increases the static (or leakage) power. The total leakage power of processor chips continued to grow for several decades.

Today, the rise of leakage power has two main implications for processor design: (1) processor frequencies are no longer increasing, (2) limiting the power consumed by processors is a first-order concern. To continue taking advantage of Moore's law and

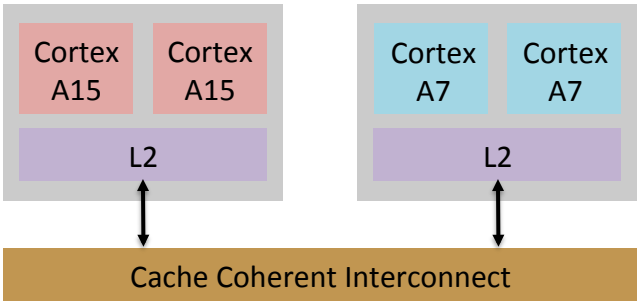


Figure 2.1: ARM's big.LITTLE processor architecture. Cortex A15 is big out-of-order core. A7 is LITTLE in-order core.

without any frequency increases, processor manufacturers have turned to multicore processors. Each multicore processor chip consists of two or more (uniprocessor) core. Software must expose thread-level parallelism to take advantage of multicore processors.

Manufacturers employ circuit and architecture techniques to limit the power consumed by processors during their design. During production, manufacturers have introduced methods that expose a *choice* to the software to restrict power.

2.1.2 Heterogeneous Multicores

Early multicore processors were homogeneous. All cores had the same architectural capabilities, i.e., superscalar width, instruction issuing heuristic, and the type of branch predictor. These similar cores ran instructions out of the program order and exploit the instruction-level parallelism (ILP) and memory-level parallelism (MLP) exposed by the software. However, the degree of ILP and MLP in applications varies. At one extreme are applications that exhibit no ILP or MLP (e.g., pointer chasing code). The powerful cores on a homogeneous multicore consume much power which is wasteful for applications that do not expose sufficient ILP or MLP.

Heterogeneous multicores have emerged because of the need for energy-efficient computing [76, 77]. Industry examples of single-ISA heterogeneous multicores include ARM's big.LITTLE [50], NVidia's Tegra [96], and Intel's QuickIA [30]. These systems contain a mix of cores that vary in their ability to exploit instruction-level parallelism (ILP) and memory-level parallelism (MLP). Big cores that run instructions out-of-order exploit ILP and MLP by having many instructions in flight at the same time, usually achieving the best performance. Small cores that execute instructions in order provide a low-power alternative and are limited in the amount of ILP and MLP that they can exploit. Heterogeneity provides a power versus performance tradeoff, giving the ability to select the core that best matches the software's characteristics, within performance and energy constraints. The challenge for system software is to take advantage of core heterogeneity by dynamically scheduling diverse workloads.

Even cores with similar architectural capabilities have some kind of heterogeneity. Each core has a dynamic voltage and frequency (DVFS) domain. DVFS allows to simultaneously change the core's voltage and frequency during the execution of an

application. The dynamic power P of a processor core is given by the equation: $P = ACV^2f$, where A takes into account the transistor's switching activity, C is the load capacitance, V is the supply voltage, and f is the operating frequency. Note that if we reduce the supply voltage by a factor x, we obtain a quadratic reduction of x^2 in dynamic power consumption. Furthermore, a decrease in supply voltage slows the transistor switching capability and requires a reduction in frequency. Together, DVFS enables a cubic decrease in the dynamic power consumed by the processor cores.

Scaling the voltage and frequency down slows down the core. This slowing down of the core does not impact performance if the core is waiting for memory accesses to resolve. On the other hand, compute-bound applications (or phases of an application) experience a slowdown proportional to the reduction in frequency. Software must carefully regulate DVFS to save energy without hurting performance.

2.1.3 Hybrid Memories

Dynamic random access memory (DRAM) has served as the main memory of all computer systems for several decades. In recent times, DRAM faces two main challenges. The manufacturing complexity of DRAM is increasing. More specifically, analysts report that scaling DRAM to smaller dimensions is becoming complex [69, 58]. The result is an increase in the cost of main memory. Modern applications demand large capacities as their working set sizes keep increasing. DRAM cells also consume idle (static) power. Static power severely limits main memory scalability.

Due to the challenges facing DRAM, scientists are investigating new memory technologies to replace DRAM. Emerging non-volatile memory (NVM) technologies, e.g., phase-change memory (PCM), spin-torque transfer (STT) RAM, magnetoresistive RAM (MRAM), and ferroelectric RAM (FRAM) promise better scalability (capacity) than DRAM. Of these emerging technologies, PCM is the most promising. PCM has several advantages: (1) it offers more capacity than DRAM, (2) it consumes no idle power, (3) it is byte-addressable, and (4) it is persistent. Despite its several advantages, PCM alone cannot replace DRAM. The reason is that PCM has two main disadvantages: (1) latency is high, and (2) write endurance is low.

Write endurance of PCM is challenging because writes to PCM cells change their material form. PCM is a resistive memory technology. PCM cells store information as the change in the resistance of the resistive material. Chalcogenide glass is the material used in several existing PCM prototypes. Chalcogenide can exist in two states: amorphous and crystalline. The two states have different resistance values. Furthermore, heating the chalcogenide switches the material between the two states.

Hybrid memory combines DRAM and PCM to offer the best of both technologies. When appropriately managed, hybrid memory offers byte-addressability, high endurance, low idle power, and low latency. There exist many prior works that use hardware and OS approaches to manage hybrid memory.

PCM can be integrated into the storage hierarchy in different ways owing to its byte-addressable and persistent features. In this thesis, we assume PCM is part of the main memory system and place it next to DRAM. Secondary disk storage backs the data in DRAM and PCM. Figure 2.2 illustrates our assumed memory and storage

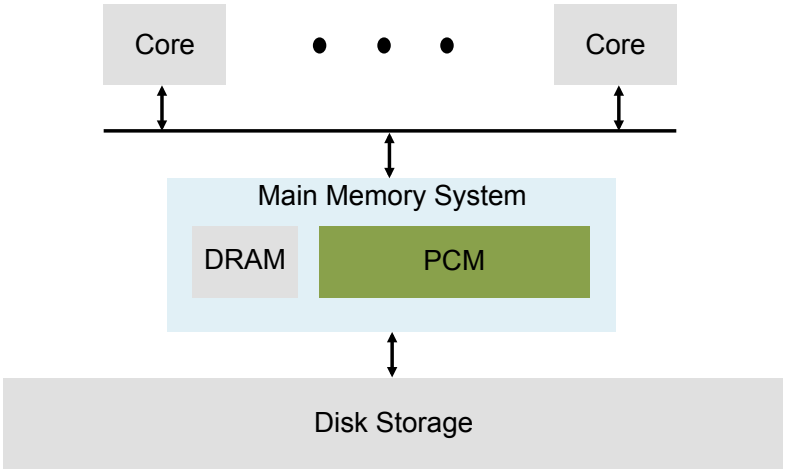


Figure 2.2: Our assumed memory and storage hierarchy.

hierarchy. Using PCM as main memory increases the capacity of the main memory system in the face of DRAM scaling challenges.

2.2 Software

2.2.1 Managed Languages and Runtimes

According to the TIOBE index for February 2019, managed languages such as Java, C#, Python, JavaScript, Scala, Ruby, and PHP are the most popular languages among programmers today [119]. Managed languages enable platform independence. More specifically, the compiler produces bytecode for an abstract virtual machine. Besides, these languages offer a range of services to the programmer. A valuable service is garbage collection that automatically manages heap memory. Garbage collection relives the programmer from the burden of explicitly freeing heap memory. Another added advantage of managed languages is improved memory security through zero initialization. Dynamic or just-in-time compilation from bytecode to machine code is another essential feature of managed languages. A managed runtime environment encapsulates all the services offered by managed languages.

This thesis shows how managed runtimes can help exploit modern heterogeneous hardware better. We use the Java Virtual Machine (JVM) for the popular Java programming language to evaluate our ideas. We believe our ideas generalize to other managed runtimes.

The mutator and the collector. Managed languages use garbage-collected heaps. The mutator executes application code, allocating new objects on the heap. The mutator updates (mutates) the object graph, changing the destination of reference fields in objects. These reference fields are contained in heap objects, thread stacks, and static variables. Due to reference updates and mutation of the object graph, objects become

disconnected (unreachable) from the heap. The purpose of the garbage collector is to find unreachable objects, free the memory these unreachable objects occupy, and make it available to the mutator for fresh allocation. Next, we discuss garbage collection in more detail.

2.2.2 Garbage Collection

The memory manager, or garbage collector (GC), provides regions of free memory to the application for fresh allocation, and automatically detects unused parts of memory and reclaims them to be used again. In this thesis, we are concerned with tracing garbage collection.

A tracing collector determines which objects should be deallocated by tracing the reachability graph of the heap. The collector first identifies *roots* from which to trace. Roots are objects directly accessible to the mutator without going through other objects on the heap. Roots include static and global variables, thread stacks, and variables stored in processor registers.

A tracing collector maintains a list of addresses to be traced. Initially, the roots are added to the list. The collector threads process the list by following pointers that point into the heap. When a heap object is found, it is marked as "live", and is searched for pointers to other heap objects, which are added to the list. manner. Tracing is complete when there are no more elements on the processing list. All heap objects that are then not marked as "live" are unreachable by the application, and thus are freed to be used again.

Garbage collection has a space-time tradeoff. The more heap space you give the application, the longer until a collection must be invoked. However, when the collection does occur because the heap is full, it could have more memory to trace, depending on the lifetime of objects, and thus take longer. On the other hand, if heap space is limited, garbage collection must be performed frequently, and yet, each collection does not last as long because the amount of live data is bounded.

Because garbage collection must look at all pointers to heap objects, it must see a consistent view of the heap. Thus, the easiest way to implement a garbage collector is by stopping the application completely during tracing and freeing. This is called *stop-the-world (STW)* collection. However, stopping the application completely while the whole heap is scanned causes long application pauses, which are undesirable, especially for interactive and real-time applications.

2.2.3 Generational Garbage Collection.

High-performance collectors today exploit the generational hypothesis that many objects die young [120]. The application (*mutator*) allocates new objects contiguously into a nursery. When allocation exhausts the nursery memory, a *minor* collection first identifies live *roots* that point into the nursery, e.g., from global variables, the stack, registers, and the mature space. It then identifies *reachable* objects by tracing

references from these roots. It copies reachable objects to a mature space and reclaims all nursery memory for subsequent fresh allocation.

2.2.4 GenImmix

In this thesis, we modify GenImmix, the default best-performing collector in Jikes [18], to create Kingsguard collectors. GenImmix uses a copying nursery and a *mark-region* mature space. The mark-region mature space consists of a hierarchy of two regions: blocks and lines. Blocks are multiples of page sizes and consist of multiple lines. Lines are multiples of cache line sizes. Objects may cross lines, but not blocks. Bump pointer object allocation is contiguous in the nursery. (Contiguous allocation is known to outperform free-list allocators due to its locality benefits [12, 18, 61].) Filling the nursery triggers a collection, which copies nursery survivors contiguously into free lines within blocks in the mature space. Filling the mature space triggers a full heap collection. Immix reclaims the mature space at a line and block granularity by marking lines and blocks live as it traces and marks live objects. Subsequent mature allocation bump-point allocates first into contiguous free lines in partially free blocks and then into completely free blocks. Allocation and reclamation use per-thread allocators and work queues to deliver concurrency and scalability. The per-thread allocators obtain blocks (partially and completely free) from a global allocator.

We use the default settings for Immix, including the maximum object size (8 KB), line size (256 bytes), and block size (32 KB). These settings match the Immix line size to the PCM line size. Immix tailors the heap representation to match the hardware memory system for performance, but it also matches the needs of PCM memory management for detecting and tolerating line failures, as Gao et al. [48] show.

2.2.5 Java Virtual Machine

In this thesis, we use the open-source Jikes Research VM (RVM) 3.1.2 as our platform because it combines good performance with software engineering advances that make it easy to modify [5, 46]. Jikes RVM is a Java-in-Java VM with a baseline compiler (no interpreter), just-in-time optimizing compiler of hot code, and a large number of state-of-the art garbage collectors [13, 12, 18, 111]. It also offers a wide range of easy-to-change barriers. In particular, we use write barriers, which call a specially-defined method on all writes to do bookkeeping. Reference write barriers are widely used in garbage collectors to track pointer references between independently collected regions [128]. We modify them to profile object writes to values as well as references. A clean interface between the compiler and collector [46] defines object layout, references, interior references, and object metadata in a few places. In contrast, changing barriers, object layout, or metadata in the widely-deployed Hotspot system [97, 98] requires numerous wide-ranging changes in the compiler and garbage collection code.

2.2.6 Java Performance Evaluation

Just-in-time compilation introduces non-determinism in Java performance evaluation. In this thesis, we use replay compilation [11], current practice in rigorous Java performance evaluation, to eliminate non-determinism introduced by the compiler. During profiling runs, the optimization level of each method is recorded for the run with the lowest execution time. The JIT compiler then uses this optimization plan in our measured runs, optimizing to the correct level the first time it sees each method [11, 61]. To eliminate the perturbation of the compiler, we measure results during the second invocation, which represents application steady-state behavior. In all results we report in this thesis, we run each application four times, and report the averages in the graphs.

Chapter 3

GC-Criticality-Aware Scheduling on Heterogeneous Multicores

3.1 Introduction

The managed language runtime environment offers increased software productivity and portability. One key reason why managed languages are used in a broad spectrum of domains, ranging from data centers to handheld mobile devices, is that they offer automatic memory management through garbage collection (GC). Garbage collection reduces the chance of memory leaks and other memory-related bugs, while easing programming. However, garbage collection introduces overhead to the application's execution time [24], in part because managed language applications allocate objects rapidly [12, 132].

Garbage collection can be run in either a "stop-the-world" mode, where the application's progress is stopped while collection occurs, or in a "concurrent" mode, where the application and GC run at the same time. However, concurrent collection threads must coordinate and share resources with the application. Moreover, if the allocation rate exceeds the rate of collection, the application can run out of allocation space, which requires the application to be stopped while GC frees memory. This can lead to a large performance penalty.

On the hardware side, heterogeneous multicores promise energy-efficiency. These processors contain a mix of big and small cores. Big cores execute instructions out-of-order to exploit instruction-level parallelism (ILP) and memory-level parallelism (MLP). Small cores are in-order and offer power-efficiency for memory-bound workloads. A significant body of recent work emphasizes the importance of scheduling on single-ISA heterogeneous multicores [9, 27, 49, 72, 79, 87, 88, 112, 114, 123, 122]. However, managed runtime environments include several service threads, such as garbage collection, that run for a significant fraction of the execution time [24, 40],

and should be treated differently than application threads, according to recent research [24, 53, 59, 90, 107]. Previous work argues that because GC threads are not on the critical path, are memory-bound, and do not exhibit ILP, they should be scheduled on small cores in a heterogeneous multicore for the best performance per energy [24].

In this chapter, we explore the behavior of concurrent garbage collection on big versus small cores for Java applications, aiming to optimize total application performance. Running benchmarks in the Jikes Research Virtual Machine (RVM) on top of a multicore simulator, we find that some applications, particularly multi-threaded applications with higher thread counts, are more garbage collection intensive, and benefit significantly if GC is run on big versus small cores, by as much as 18%. These benchmarks exhibit *GC criticality* during execution when the concurrent GC threads cannot keep up with application allocation, and thus GC threads must pause application progress and divert to a stop-the-world mode to collect memory. For other applications, however, we observe no performance difference when running GC threads on big versus small cores. In particular, single-threaded and some multi-threaded applications at small thread counts do not exercise GC much, and we call them *GC-uncritical*. To verify the generality of GC criticality, we also compared the performance of Jikes' best-performing production collector, stop-the-world generational Immix, when it runs on big versus small cores. Several benchmarks still benefit from running on out-of-order cores, as they demonstrate a performance difference of up to 15%. We conclude that GC criticality can occur in many different system setups. GC criticality is a function of a number of factors, including processor architecture, virtual machine, garbage collection algorithm and implementation, heap size, application characteristics, etc. The bottom line is that if garbage collection is unable to keep up with the application's memory allocation rate (because GC is receiving too few resources), garbage collection will become critical.

Based on these insights, we design a new, adaptive scheduling algorithm that responds to signals from the managed language runtime about GC criticality, which dynamically varies during the run, boosting GC threads' priority on the big core(s) only if GC is in danger of not keeping up with application allocation. Our GC-criticality-aware scheduler adapts to phase behavior, balancing performance and energy efficiency by lowering GC threads' priority on the big core(s) if GC becomes uncritical. While our scheduler is performance-neutral for GC-uncritical benchmarks, it improves performance significantly for GC-critical applications (compared to prior best practice which puts GC threads always on small cores [24]). Using a set of Java benchmarks from the DaCapo benchmark suite [14] on top of the Jikes Research Virtual Machine 3.1.2 [4], we report an average performance improvement of 2.9%, 7.8%, and 16% for the GC-critical benchmarks when running on a four-core system with one, two, and three big cores, respectively, while at the same time improving energy-efficiency by 3.5%, 10.7% and 20%. Compared to an existing fair scheduler [122] which strives at achieving fairness across all runnable threads, our GC-criticality-aware scheduler achieves significantly better performance, especially for architectures with limited big core resources. We comprehensively evaluate the robustness of GC-criticality-aware scheduling across core counts, big to small core ratios, heap sizes, and clock frequency settings, and conclude that GC-criticality-aware scheduling is particularly beneficial as GC becomes more critical. GC-criticality-aware scheduling improves

overall application performance by giving sufficient resources to GC so it can keep up with the application.

This chapter makes the following contributions:

- We demonstrate, contrary to prior work, that garbage collection can significantly benefit (up to 18%) from out-of-order versus in-order execution by exploiting ILP.
- We pinpoint when GC becomes critical to overall application performance, namely when a concurrent collector cannot free memory fast enough for application allocation.
- Motivated by the observation that applications exhibit different sensitivities with respect to GC criticality, we propose an adaptive scheduling algorithm that receives semantic information from the memory manager about GC criticality, adjusting GC's priority for big core time slices, even taking slices away from the application so as to avoid costly stop-the-world pauses.
- We evaluate our adaptive scheduling algorithm, showing that it performs well across a large range of heterogeneous architectures and heap sizes. While our GC-aware scheduler is performance and energy-neutral for GC-uncritical applications, we see substantial performance and energy efficiency improvements for GC-critical applications.

This work shows that scheduling modern workloads on heterogeneous multicores significantly benefits from semantic information (GC criticality) provided by the managed runtime, in order to provide high performance on future energy-efficient processor architectures.

3.2 Related Work

We now describe work related to scheduling multithreaded and managed workloads on (heterogeneous) multicores.

3.2.1 Scheduling Managed Language Workloads on Heterogeneous Multicores

A number of prior works evaluate how best to schedule managed language workloads on heterogeneous multicores, suggesting that it is best to put service threads on small(er) cores. Recent work characterizes service threads in the Oracle HotSpot JDK 1.6.0 and Jikes RVM, and explores the opportunity to isolate service threads to small cores to optimize performance per energy [24]. They focus on service threads in isolation, and argue that as garbage collection runs asynchronously with the application, is not on the critical path and is memory-bound, it is better to run GC on small cores. In contrast, we focus on end-to-end performance and find that GC could end up on the critical path and hurt overall application performance if always left to execute on

the small core(s). We also achieve better energy efficiency in comparison with their policy.

Prior work also advocates to exploit simple in-order cores to run low-priority service threads in co-designed virtual machines [53]. More recent work explores changing the garbage collection algorithm to be amenable to running on a GPU [90]. Researchers have studied core adaptation for managed applications for the purposes of energy reduction [59]. They conclude that whereas the application benefits most from wide-issue out-of-order cores, GC threads prefer simpler cores, albeit still out-of-order but with a smaller instruction window. Our results are in line with this finding: GC benefits from periodically running on a big core to improve overall performance.

3.2.2 Managed Language Workloads on Multicores

Prior work explores scheduling Java workloads on modern multicore hardware to get the best performance. Sartor et al., evaluate the effect of isolating garbage collection threads to another socket, and scaling down the frequency of that socket [107]. They conclude that slowing down the clock frequency of garbage collection degrades performance, which is in line with our findings, also noting that it degrades performance much less than when scaling down application threads.

Esmaeilzadeh et al., evaluate performance and power consumption across five generations of processors, concluding that managed language workloads are more power-hungry and exploit more parallelism than native single-threaded workloads, further motivating the relevance of our work [43].

Some prior work proposes hardware support for concurrent GC, adding hardware for new ISA instructions [64], to the memory subsystem [109], to the CPU pipeline [53], or with a completely custom design [32]. In our work, we assume 'stock' heterogeneous multicores with big and small cores.

3.2.3 Scheduling on Heterogeneous Multicores

Kumar et al., advocate single-ISA heterogeneous multicores for energy efficiency reasons [76, 77]. This has spurred a flurry of related work in scheduling for heterogeneous multicores; see for example [9, 27, 49, 72, 79, 87, 88, 112, 114, 123].

Prior work proposes fairness-aware scheduling for multi-threaded (native) applications [122]. A key difference is that we acknowledge the inherent thread heterogeneity in managed language workloads, treating GC threads differently from application threads.

A number of recent works have looked into dynamically identifying and speeding up critical threads or bottlenecks [39, 40, 65, 118]. Suleman et al. accelerate critical sections by migrating threads to a big core [118]; Joao et al. generalize this concept to other bottlenecks including barriers and producer-consumer synchronization [65]. Du Bois et al. identify critical threads during multi-threaded program execution based on the number of co-executing threads per time interval as delineated by synchronization activity [39, 40]. None of these readily apply to concurrent garbage collection

in managed language workloads. Our work demonstrates that providing semantic information about GC criticality from the managed runtime helps the scheduler to give big core cycles to GC threads dynamically when needed.

3.3 Background

In Chapter 2, we provide background on managed languages and garbage collection. In this section, we first provide background on concurrent garbage collection. We then discuss garbage collection on heterogeneous multicores.

3.3.1 Concurrent Garbage Collection

Concurrent garbage collection runs garbage collection threads alongside application threads to reduce pause times. In this work, we consider the Jikes Research Virtual Machine's (RVM) [4] concurrent collector, which is a traditional mark-sweep snapshot-at-the-beginning concurrent GC algorithm, based on Yuasa's algorithm [130]. Figure 3.1 depicts the phases of an application that has four application threads (*a0* to *a3*) running with such a collector. This collector has four threads, with *g0* and *g1* running concurrently with the application, and *g2* and *g3* running when the application is stopped.

Most concurrent collectors require a small pause to the application to first identify a consistent root set (shown in Figure 3.1 as "roots"), and later to actually free memory (shown as "release"). In our concurrent collector, separate threads are spawned to perform the STW phases of collection (threads *g2* and *g3*). The traversal of the object graph can happen in parallel with the application (shown as the action of threads *g0* and *g1*) as long as newly allocated or modified objects are marked as "live" so that they are not freed by the collector. In addition, all application writes go through a barrier to coordinate with GC threads so that they are not writing to the same object, and so the GC maintains a consistent view of heap pointers [17]. Our concurrent collector initiates a new collection cycle (defined as starting with the "roots" phase, and ending with the "release" phase) after the previous cycle ends and if a parameter-defined quantity of memory in bytes has been allocated.

While pauses of the application are minimized when using a concurrent garbage collector, the application execution can still be stalled. If the application runs out of memory to allocate into, it must pause until garbage collection frees up enough memory for it. Jikes' collector then transitions into a stop-the-world mode (shown on the right in Figure 3.1 as the "scan" phase that makes collection slower). This STW pause can have a large performance cost, especially because bookkeeping work must be performed to transition from the concurrent to the STW mode, and switching threads could also cause cache perturbation.

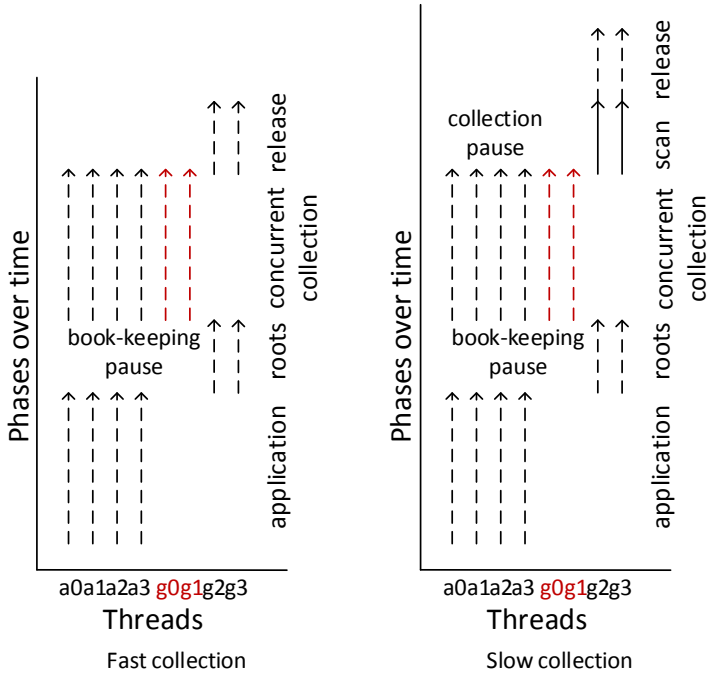


Figure 3.1: Threads and phases of application execution using Jikes' concurrent collector, with the optional 'scan' pause (right) if concurrent GC threads cannot keep up with application allocation.

3.3.2 Garbage Collection on Heterogeneous Multicores

While some prior work [24, 59] has explored the behavior of managed language services, including garbage collection, on heterogeneous cores, they have focused on optimizing energy. They found that GC can be put on a smaller core, or a scaled-down big core, in order to save energy. Both prior works argue that GC does not have instruction-level parallelism, and uses a lot of memory bandwidth. Another work [107] explored separating GC threads to another socket and scaling down the frequency, revealing that when GC threads in particular are scaled down, there is an overall increase in execution time.

In this chapter, we focus on minimizing the execution time of managed language applications running on a heterogeneous multicore through scheduling. If garbage collection is performed in STW mode, it is obvious that it is critical (i.e., holding up the progress of the application), and thus should be transferred to the big core, even if the heterogeneous system has limited big core resources. However, the problem is more complex with a concurrent collector that runs alongside the application, which has to coordinate during allocations and writes to references. Furthermore, the GC and application compete for core and memory resources. Of course the application's progress is most critical; however, if GC has to stop the application to finish scanning the heap, it is on the critical path. The criticality of concurrent GC depends on how fast the application is using memory (including its allocation rate and object sizes and

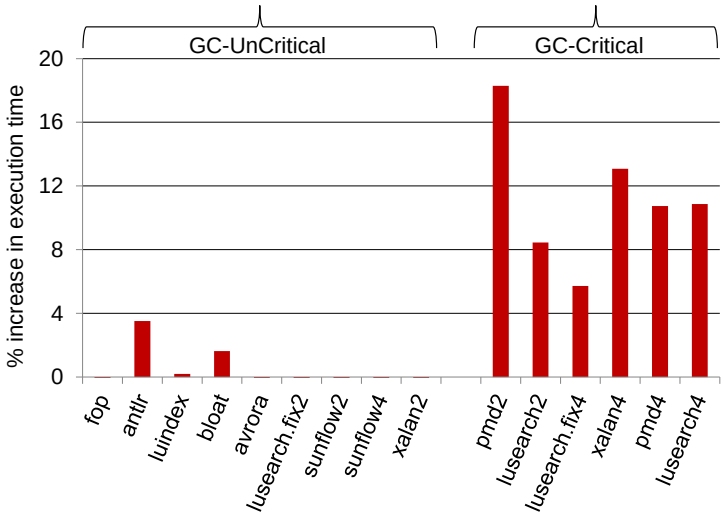


Figure 3.2: Total execution time increase when concurrent GC threads are run on small versus big cores, with each other thread always pinned to its own big core. *Four multi-threaded applications are GC-critical, while the others are GC-uncritical.*

lifetimes), and how fast the collector is able to free up memory. We aim to design a scheduler that responds to GC criticality by receiving hints from the managed runtime, dynamically adapting the GC's share of big core cycles to achieve the best application performance.

3.4 Concurrent GC on Heterogeneous Multicores

Before presenting our adaptive scheduling algorithm, we first explore the behavior of concurrent GC threads on heterogeneous multicores in more detail, to further motivate the need for an improved scheduling algorithm and to indicate the potential of heterogeneity. In our first experiment, we assess the behavior of concurrent garbage collection threads running on different core types, small versus big. We use two GC threads (as mentioned in Section 3.3.1, this means that there are two concurrent and two STW threads); we also pin threads to cores, and set the number of cores equal to the number of threads. To assess GC's behavior on the different core types, we compare a run using eight big (out-of-order) cores for all threads to a run using six big cores for all non-GC and STW GC threads and two small (in-order) cores for the two concurrent GC threads (see Section 3.6 for more methodological details).

Some multi-threaded benchmarks exhibit *GC criticality*, while other benchmarks do not. Figure 3.2 shows the percentage increase in total application execution time when concurrent GC threads are run on small versus big cores, normalized to when all GC threads are on the big cores. Figure 3.2 shows that the execution time difference can go up to 18% for pmd2. All but one four-threaded application, and two two-threaded applications, have a large difference in execution time, which corresponds

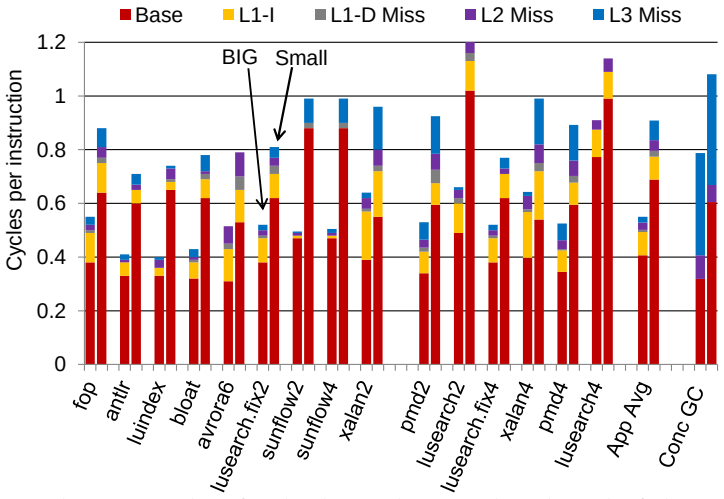


Figure 3.3: The CPI stacks of only the application threads and of the concurrent garbage collector (rightmost bars) when run on big (left bar) and small (right bar) cores. *Concurrent GC threads exploit more ILP on the big core, nearly halving the CPI base component.*

with an increase in the time spent in stop-the-world mode. On the other hand, most single-threaded, and few multi-threaded, benchmarks (antlr, bloat, fop, luindex, avrora, lusearch-fix2, sunflow2, xalan2, and sunflow4) see no execution time difference. antlr sees a small execution time change because the concurrent GC time has grown. We find that avrora and sunflow, despite having many application threads, are compute-intensive, and do not spend much time performing garbage collection. We call these nine left-most benchmarks *GC-uncritical*. The six right-most benchmarks: pmd2, lusearch2, lusearch-fix4, xalan4, pmd4, and lusearch4, have a large execution time difference when concurrent GC threads run on the small versus big cores; i.e., they exhibit *GC criticality* during execution.

Scheduling concurrent garbage collection on small cores slows down GC-critical benchmarks. The large difference in execution time for the GC-critical benchmarks when concurrent GC threads run on small cores is due to longer stop-the-world pauses. These longer pauses are due to more optional "scan" pauses, as shown on the right in Figure 3.1. We find that other STW phases, "roots" and "release" are relatively short on average. What increases execution time substantially is when the concurrent collection threads cannot scan and free memory in time before an application allocation request fails, and the world must be stopped for GC threads to finish scanning the heap. This is more likely to happen in multi-threaded benchmarks, where many threads are rapidly allocating memory, which increases the amount of GC work and time [40]. Avoiding the critical and crippling STW "scan" phases is key to improving GC, and therefore overall application performance.

Concurrent garbage collection exploits ILP on the big core. To better understand why concurrent GC benefits from running on a big core for the GC-critical benchmarks, we present the CPI stacks [45] for the application threads versus the concurrent garbage collection threads in Figure 3.3 when running on big versus small cores.

To isolate application threads' behavior on the different core types, we run these threads on all big versus all small cores. Each stack shows the base component, representing committed instructions and useful work done, and the memory components, including time waiting for cache and memory accesses. The total cycles per instruction is the sum of the base and the memory components. We find that the concurrent GC threads (rightmost bars called "Conc GC") benefit substantially from running on the big out-of-order versus the small in-order core, with the benefit coming primarily from a substantial reduction in the base component. This suggests that the out-of-order core is able to exploit instruction-level parallelism (ILP) in the concurrent GC threads, hiding instruction latencies and inter-instruction register dependencies. While the collector stacks show a large memory component, larger than that of our applications, we observe there is limited memory-level parallelism (MLP), as there is little change in the memory component between the big and small core runs.

3.4.1 Generalizing to Different Garbage Collectors

We want to demonstrate that GC criticality is not just a function of the GC algorithm we are using. Thus, we perform experiments analyzing the behavior of Jikes RVM's best-performing production collector on both big and small cores to show that GC in general can benefit from out-of-order processing. Immix is a generational, stop-the-world collector. We run the Immix collector with two threads pinned to two separate cores on the Sniper simulator, and other experimental setup details (such as heap size) are the same as in the concurrent GC experiment. We always place application threads on out-of-order cores. Figure 3.4 shows the percentage increase in total execution time from running the Immix collector threads on in-order versus out-of-order cores. The benchmarks identified as GC-critical when running with the concurrent collector (on the right) also see an increase in execution time when Immix runs on the small cores: up to 15%. We also see a large execution difference for xalan2 with the stop-the-world collector. The overall conclusion is that garbage collection exhibits ILP and thus benefits from running on a big out-of-order core in a heterogeneous multicore machine.

To further show that GC criticality exists in other environmental setups as well, we perform experiments with another JVM, OpenJDK. We run the DaCapo benchmarks with OpenJDK's concurrent collector on real hardware, and use frequency scaling. The results show that the same benchmarks that exhibit GC criticality with Jikes also show GC criticality with OpenJDK.

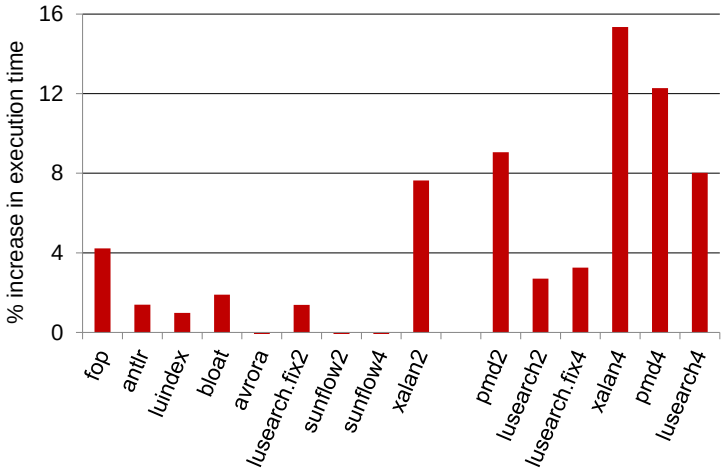


Figure 3.4: Total execution time increase when STW GC threads are run on small versus big cores, with each other thread always pinned to its own big core. *The production garbage collector also benefits from out-of-order execution.*

3.5 GC-criticality-aware Scheduling

Our adaptive scheduling algorithm measures GC criticality during run-time and dynamically adjusts the GC's priority to run on the big core(s) based on feedback about STW pauses, particularly detrimental scan pauses, incurred by the concurrent collector. The algorithm is reactive, but tries to keep GC threads on the small core(s) when GC is not critical, to let application threads use the big core cycles, while sharing big core time slices between the GC and application threads when GC is critical.

3.5.1 Base Schedulers

Before describing the GC-criticality-aware scheduler in more detail, we first revisit previously proposed schedulers on which we build and to which we compare. We discuss two such schedulers: one called *gc-on-small* that keeps concurrent GC threads on small core(s), and a second we call *gc-fair* that gives all threads equal time on the big core(s). The first scheduling approach, *gc-on-small*, is patterned after the recommendations of previous research to always put GC threads on small cores for better energy usage [24]. We use this previously proposed scheduler as our baseline. The second, *gc-fair*, uses the algorithm proposed in [122], which was devised for native multi-threaded workloads, and was not previously evaluated for managed language workloads. This scheduler gives all runnable threads an equal percentage of time on the big core in a round-robin manner. Each time slice, the thread with the least cumulative big core time is picked to move to a big core. This implies that with four application threads, *gc-fair* would give two GC threads 33% of time slices on big cores, whereas with two application threads, it would give 50%, and with one, 66%.

The two base schedulers are graphically depicted in Figure 3.5. Each row of boxes shows a different scheduler, and the columns depict different four-core heterogeneous

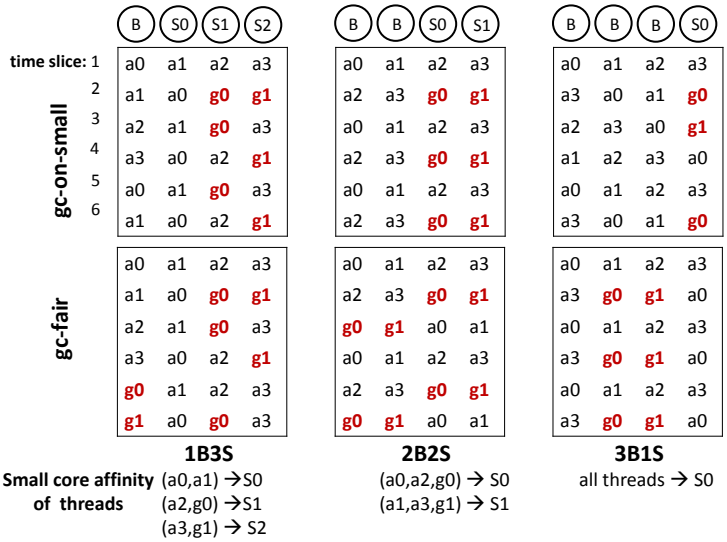


Figure 3.5: Picture depicting two schedulers across heterogeneous architectures using four application and two GC threads.

architectures. We denote the architecture of a heterogeneous multicore as mBnS, with m big cores and n small cores. We vary the number of big cores across these heterogeneous configurations: 1B3S, 2B2S, 3B1S. The contents of each box then shows which thread would be scheduled on which core, showing scheduling decisions for the first six time slices. We consider four application threads and two GC threads in this figure. These algorithms only change the scheduling of the concurrent GC threads (i.e., *g0* and *g1* from Figure 3.1), as we always put STW GC threads on the big core(s) because the application is no longer running. The bottom of Figure 3.5 also depicts that in these base schedulers, each thread has an affinity to a particular small core to exploit locality. However, the schedulers will schedule a thread waiting to run on any available idle core.

The base schedulers both have limitations. *gc-on-small* keeps the concurrent GC threads on the small core(s), which may lead to substantial performance losses for GC-critical applications. *gc-fair*, on the other hand, takes away big core cycles from the application thread(s) when scheduling GC on the big core(s), which may be detrimental for GC-uncritical applications.

3.5.2 GC-criticality-aware Scheduler

Developing a scheduling algorithm for concurrent GC on a heterogeneous multicore is not trivial. GC criticality is not only a function of the application and system architecture, including the number of cores, ratio of big to small cores, clock frequencies, etc. It is also a function of the GC algorithm and heap size. GC criticality is a dynamic characteristic that is based on the application's allocation speed versus the collection speed. An application becomes GC-critical if its threads progress faster, thus allocating objects faster and needing GC to collect memory faster. Thus, statically determining

ALGORITHM 1: Our GC-criticality-aware scheduler. T_s is the sampling interval. I_{max} is the threshold for the number of consecutive intervals when GC is observed not to be critical before degrading to a *gc-on-small* scheduler.

```
input T_s, I_max
:
initial scheduler is gc-on-small;
noise-margin = 100 micro seconds;
while true do
    wait for an STW pause;
    start a new sampling interval;
    T_scan = sum_{i=1}^{n} T_scan(i);
    end of a sampling interval;
    if scheduler is gc-on-small then
        if T_scan ≥ noise-margin then
            new scheduler is gc-boost;
        end
    end
    if scheduler is gc-boost then
        if T_scan ≥ noise-margin then
            zero-scan-intervals = 0;
            upgradeGCBoostState();
        end
        if T_scan < noise-margin then
            degradeGCBoostState();
            zero-scan-intervals ++;
            if zero-scan-intervals = I_max then
                new scheduler is gc-on-small;
            end
        end
    end
end
```

GC criticality for a particular application run, and choosing between *gc-on-small* and *gc-fair* is not enough. We need an adaptive GC-criticality-aware scheduling algorithm that is robust across system architectures and workload execution variations.

The fundamental principle and key insight of our adaptive scheduling algorithm is to schedule collector threads on small cores unless GC is currently critical to the application's progress; if GC is critical, we give GC threads some big core cycles, and if it remains critical, we continue to give GC more big core cycles so that it can keep up with the application and does not need to stall to clean up memory during a long stop-the-world pause. Our dynamic algorithm to schedule GC on heterogeneous cores is shown in Algorithm 2. We always start with the *gc-on-small* scheduler. We use the notion of a sampling interval (T_s) during which we profile the behavior of the garbage collector, measuring the crippling STW scan time in particular, which we aim to reduce with this algorithm. We then react to that in the next time interval, giving GC threads more big core cycles if they incurred STW scan time, and fewer cycles if there was none. Note that the mandatory STW pause (shown in Figure 1 as "roots"),

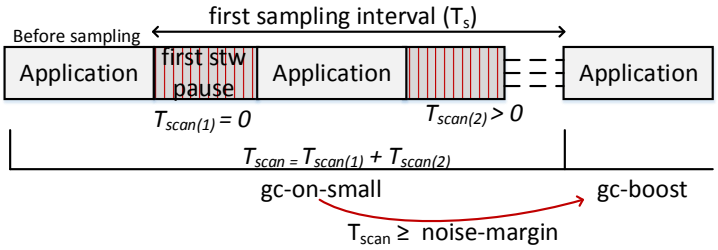


Figure 3.6: A single sampling interval in our GC-criticality-aware scheduler.

Table 3.1: GC boost states, updated/degraded by our algorithm.

State	Q(t): number of big core quanta for thread t
P0	Q(g0) = 1 ms, Q(g1) = 1 ms
P1	Q(g0) = 2 ms, Q(g1) = 1 ms
P2	Q(g0) = 2 ms, Q(g1) = 2 ms
P3	Q(g0) = 3 ms, Q(g1) = 2 ms
P4	Q(g0) = 3 ms, Q(g1) = 3 ms

marks the beginning of a new sampling interval.

A single sampling interval is shown in Figure 3.6, and each interval begins when the application encounters an STW pause. The managed runtime's memory manager communicates the beginning and end of any STW scan pause to the scheduler (i.e., the extra solid lines for threads *g2* and *g3* on the right in Figure 3.1). The scanning pause is only encountered when the concurrent GC could not keep up with application allocation, indicating GC criticality. During a particular sampling interval, we sum up all the STW scan pauses (T_{scan}). If T_{scan} is greater than a *noise-margin* (100μs), the scheduler switches from *gc-on-small* to *gc-boost* scheduling. Initially, *gc-boost* gives the GC threads equal priority with the application threads to run on the big core(s), effectively being the same as *gc-fair*. If, in subsequent sampling intervals, scan time continues to be significant, we *further increase* the priority of the GC threads, giving them even more time slices on the big core(s), thus implicitly slowing some application threads. If, on the other hand, in subsequent intervals scan time is zero, i.e., no GC criticality, we decrease GC's big core priority to give more time slices to application threads. If no scan time is observed for several intervals, we put GC threads back to run only on small cores. In this way, we continuously profile the garbage collector and update our scheduling policy, adapting to application phase behavior.

We regulate GC thread priority using the states depicted in Table 3.1. Our scheduling time slice is 1 *ms*, and thus when initially transitioning to *gc-boost* scheduling, we are in state *P*0 where each GC thread gets a 1 *ms* time quantum on the big core in a round-robin fashion with application threads. When Algorithm 2 calls *upgradeGCBoostState* to boost GC threads' priority, our algorithm goes up one state, giving one GC thread yet another big core time slice. We allow GC threads to go

Table 3.2: Simulated system parameters.

Component	Parameters
Processor	1 socket, 4 cores, 2.66 GHz
Big core	4-issue, out-of-order, 128-entry ROB
Small core	4-issue in-order, stall-on-use
Cache hierarchy	Private L1-I/L1-D/L2, Shared L3
	Capacity: 32 KB/32 KB/256 KB/16 MB
	Latency : 2/2/11/40 cycles
	Set-associativity: 4/8/8/16
	64 B lines, LRU replacement
Coherence protocol	MESI
Memory controller	FR-FCFS scheduling
	30.4 GB/s bandwidth to DRAM

up to state $P4$ which gives each GC thread $3\,ms$ on a big core when it is scheduled there. Similarly, if our algorithm discovers insignificant scan time, or that GC is not critical, it calls $degradeGCBoostState$, which decreases GC threads' priority on the big core(s) by going down one state. We also provide a counter, *zero-scan-intervals*, that is incremented every consecutive interval we see no GC criticality, and if it reaches a certain threshold, I_{max}, we switch back to the *gc-on-small* scheduler, to maintain energy efficiency.

3.6 Experimental Setup

For evaluating GC-criticality-aware scheduling, we use the experimental setup outlined in this section.

Simulator We perform our experiments on a simulator to evaluate scheduling algorithms across a range of potential heterogeneous architectures more easily. We use Sniper version 4.0, a parallel, high-speed and cycle-level x86 simulator for multicore systems [25]. We use the most detailed core model in Sniper. Because Sniper is a user-level simulator, it was extended by [108] to correctly run a managed language runtime environment including dynamic compilation, and emulation of frequently used system calls. Java applications are run to completion in our experiments. For reported energy results, we use McPAT version 1.0 [54] with a 22nm technology node.

Processor architecture We simulate a single-ISA heterogeneous multicore processor consisting of big out-of-order and small in-order cores, as described in Table 3.2. We set both core types to run at the same clock frequency, although we explore a small core with reduced frequency in Section 3.7.3. The core types also have the same cache hierarchy, i.e., each core has private L1 (32 KB) and L2 (256 KB) caches; the last-level L3 cache (16 MB) is shared among all cores. Most of our experiments set the total

Table 3.3: The heap sizes we use for running our benchmarks from the DaCapo suite and execution time with Sniper using a big core per thread.

Benchmark	Heap size [MB]	Execution time (ms)
avrora6	98	1,250
luindex	44	499
fop	80	322
antlr	48	811
pmd4	98	1,150
pmd2	98	950
sunflow2	108	5,917
sunflow4	108	3,065
bloat	66	4,633
xalan2	108	1,793
lusearch.fix2	68	1,779
xalan4	108	1,136
lusearch.fix4	68	1,134
lusearch4	68	4,431
lusearch2	68	4,878

number of cores to four, and we vary the ratio of big and small cores, exploring the following configurations: 1B3S, 2B2S, 3B1S. We explore configurations with a total of six cores in Section 3.7.4.

JVM-scheduler communication Our main contribution is a dynamic scheduler for heterogeneous processors that reacts to signals from the software's memory manager about GC criticality. Therefore, we modify Jikes' garbage collector to send signals to the simulator, as done by [108], using a *magic* instruction. During execution, the memory manager sends signals when STW GC threads start or stop, and in particular when they perform scanning. The thread scheduler, which is implemented in the simulator, then adapts its schedule. In all of our schedulers we use a time slice of one millisecond. The overhead of migrating threads between cores is accounted for, including restoring architecture state (we use a fixed cost of 1000 cycles), plus cache warming effects.

Java workloads We evaluate ten Java benchmarks from the DaCapo suite [14] that we can get to work with Jikes RVM 3.1.2 [4]. We use six benchmarks from the DaCapo-9.12-bach benchmark suite (avrora, luindex, lusearch, pmd, sunflow, xalan) and three benchmarks from the DaCapo 2006 benchmark suite (antlr, bloat, and fop). We also use an updated version of lusearch, called lusearch-fix (described by [129]), that eliminates

needless allocation. Four benchmarks, antlr, bloat, fop, and luindex, are single-threaded while the remaining benchmarks are multi-threaded. The avrora benchmark uses a fixed number (six) of application threads, but has limited parallelism [40]. For the remaining multi-threaded benchmarks, we perform evaluation with two and four application threads, resulting in a total of fifteen workloads (we place the number of application threads after the benchmark's name). We vary the number of threads to explore the space because, as mentioned by [40], the amount of GC work and time increases as the number of application threads increase, and thus, GC can become more critical. Table 3.3 lists our benchmarks, the heap size we use in experiments (reflecting moderate, reasonable heap pressure [108]), and their running time when using Sniper with one big core per thread.

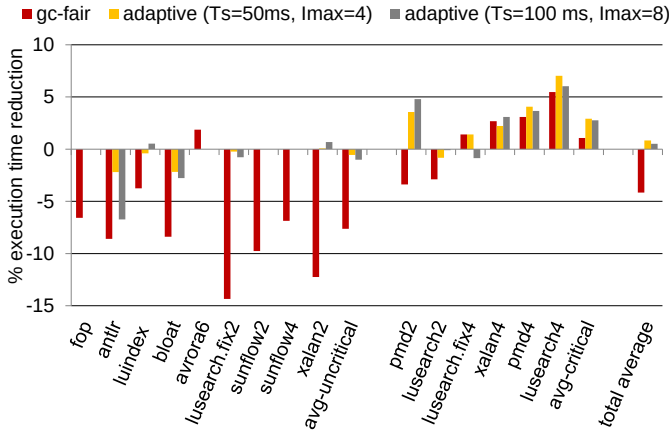
Concurrent collector The concurrent garbage collector in Jikes RVM is an implementation of the mark-sweep snapshot-at-the-beginning algorithm described by [130]. Jikes' concurrent collector runs with n stop-the-world threads and n concurrent threads, where n is a command-line parameter. In our work, we set $n = 2$, as previous research [40] shows that Jikes performs best with two GC threads, even with single-threaded benchmarks, and it does not scale well with GC thread counts above two. Because the application is not running during STW mode, in this work we always schedule the STW GC threads on the big core(s). If there is only one big core, we schedule one STW thread on the big core and leave the other on the small core because GC is a work-stealing algorithm. We use a default heap size specified in Table 3.3, which we vary in a sensitivity study in Section 3.7.5. Unless otherwise stated, a concurrent GC cycle is triggered to begin after every 8 MB of allocation (after the previous cycle finishes).

3.7 Experimental Evaluation

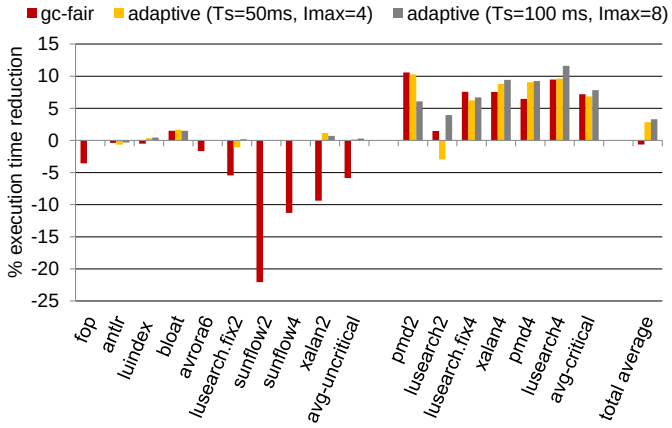
We now evaluate our GC-criticality-aware scheduler in terms of performance and energy-efficiency, across a range of heterogeneous multicore processors and configurations.

3.7.1 Performance

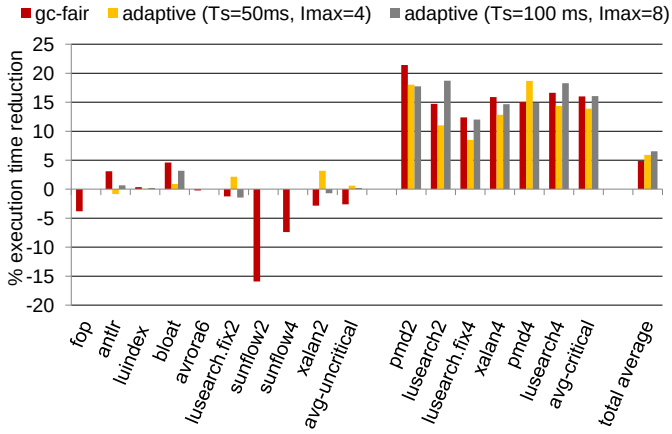
Figure 3.7 presents per-benchmark performance results for our adaptive, GC-criticality-aware scheduler on three heterogeneous architectures, also comparing to the execution time reduction of the *gc-fair* scheduler. In all results, we normalize to when GC is run on small cores (*gc-on-small*). The graphs present our adaptive scheduler results using different algorithm parameter configurations, with a sampling interval of $100\,ms$ and I_{max} of 8, as well as an interval of $50\,ms$ and I_{max} of 4. Our graphs present averages per category: for benchmarks identified as GC-uncritical, GC-critical, and then a total average across all benchmarks.



(a) 1B3S



(b) 2B2S



(c) 3B1S

Figure 3.7: Execution time reduction (%) of adaptive and *gc-fair* schedulers compared to *gc-on-small* on three heterogeneous architectures. *Our adaptive scheduler is robust across architectures.*

GC-criticality-aware scheduling performs well across heterogeneous architectures, greatly improving the performance of GC-critical applications over *gc-on-small* Looking at individual benchmark trends in Figure 3.7, we see the same six benchmarks that Figure 3.2 identified as GC-critical as those that benefit most from our adaptive scheduling algorithm. We see a clear trend that performance gains increase as we add more big cores. For the three heterogeneous architectures, 1B3S, 2B2S and 3B3S, we observe an average performance improvement of 2.9%, 7.8%, and 16%, respectively, for the GC-critical benchmarks. The reason for this increase is that with more big cores, application threads run, and thus allocate, faster. GC becomes more critical, hence boosting the priority of GC through our GC-criticality-aware scheduler becomes more beneficial.

GC-criticality-aware scheduling improves over *gc-fair* for heterogeneous multi-cores with limited big core resources On a 1B3S architecture, *gc-fair* severely degrades performance for GC-uncritical benchmarks, whereas our adaptive scheduler is on-average performance-neutral. Furthermore, for GC-critical benchmarks, our algorithm generally sees slightly higher performance improvements than *gc-fair*, with some results being similar or slightly lower due to the reactive nature of our scheduler. We see larger reductions in execution time when our algorithm responds to phase behavior about GC criticality and boosts the number of big core cycles given to GC threads over *gc-fair*.

We show one such case in Figure 3.8 for pmd4 on 1B3S. The x-axis shows each sampling interval, and the y-axis shows the portion of that 100 *ms* interval that was measured as being stop-the-world, as well as the STW time just for scanning. In the 2nd interval, STW scan time is significant, hence the scheduler switches to *gc-boost* in the next interval. As STW scan time keeps on being significant, the GC boost state (shown at the top) is increased up to $P4$, giving more big core cycles to GC threads. GC threads continue to be critical, as they cannot scan the heap fast enough to keep up with application allocation. We see that in interval 9, GC finally becomes non-critical, but again becomes critical in intervals 10 and 11. This case clearly illustrates the need for boosting GC priority beyond *gc-fair* while adapting to GC criticality.

GC-criticality-aware scheduling greatly reduces the negative impact of *gc-fair* for the GC-uncritical applications On average across all benchmarks, while *gc-fair* increases total average execution time by 4% on a 1B3S architecture, our adaptive algorithm decreases execution time slightly, by 1% — a net performance improvement of 5% over *gc-fair*. We see the same trend on a 2B2S architecture, noting that our adaptive algorithm improves performance by about 3%. On a 3B1S architecture, the *gc-fair* scheduler can improve performance by about 5% over *gc-on-small*; however, it also degrades performance severely by over 15% for sunflow2. Our adaptive algorithm, on the other hand, improves performance on average by over 6%, and is more robust across applications (with no negative outliers).

Both antlr and bloat exhibit slowdowns with our adaptive algorithm on a 1B3S architecture. For antlr, using the larger sampling interval size of $T_s = 100$ and the large degradation threshold of $I_{max} = 8$ particularly hurts performance. We explain

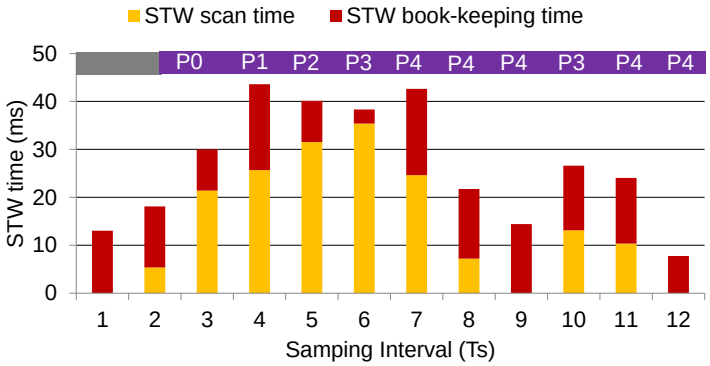


Figure 3.8: STW phase behavior over time for pmd4 on 1B3S with our adaptive scheduler, $T_s = 100ms$ and $I_{max} = 8$.

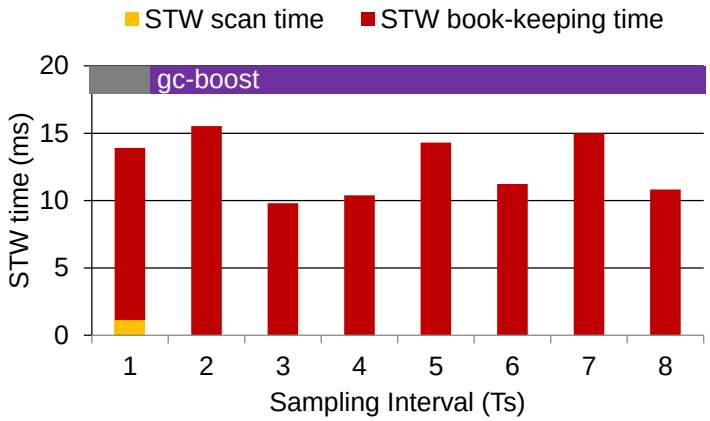


Figure 3.9: STW phase behavior over time for antlr on 1B3S with our adaptive scheduler, $T_s = 100ms$ and $I_{max} = 8$.

antlr's behavior in Figure 3.9. In antlr's 1st interval, it has a significant STW scan time, and thus our adaptive algorithm switches to the *gc-boost* scheduler. When using a large I_{max} like 8, the scheduler remains set to *gc-boost* until it sees 8 consecutive intervals where GC is not critical, meanwhile taking many big-core cycles away from the application. Because antlr is such a short-running benchmark, it never switches back to the *gc-on-small* scheduler, and thus performance is degraded because antlr is in fact GC-uncritical. Figure 3.7 shows that with more conservative values, $T_s = 50$, $I_{max} = 4$, antlr's performance degradation reduces.

GC-criticality-aware scheduling is robust to its parameter settings Figure 3.10 evaluates the impact of different parameters on our algorithm: with a sampling interval of 100 *ms* and an I_{max} of 2, 4, and 8, then setting I_{max} to 4 and using a sampling interval of 50 *ms*. We present averages across heterogeneous architectures per benchmark category, showing little performance variation across parameters. However, we find that, as shown in Figures 3.7 and 3.10, larger T_s and I_{max} values are not as beneficial for an architecture with only one big core. Big core cycles are more precious, i.e., taking

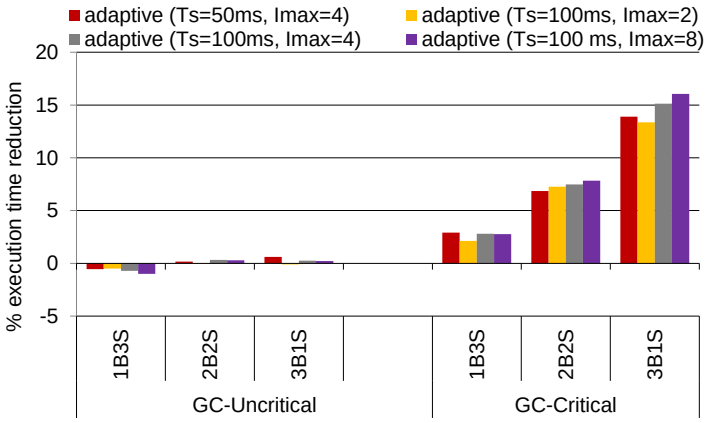


Figure 3.10: Average percentage execution time reduction across benchmarks per category for different T_s and I_{max} values. *Our GC-criticality-aware scheduler is robust across its parameter settings. Small values of T_s and I_{max} are better performing with a single big core, whereas larger T_s and I_{max} are better with more big core resources.*

them away from the application thread(s) hurts total performance more, especially for the short-running antlr benchmark (as shown in Figure 3.9). We therefore use $T_s = 50$ and $I_{max} = 4$ for our scheduler on the 1B3S architecture to more conservatively use the big core, while for all other heterogeneous configurations, we use the default of $T_s = 100$ and $I_{max} = 8$.

3.7.2 Energy Efficiency

While GC-criticality-aware scheduling improves performance, it also improves energy-efficiency for GC-critical applications. We use the energy-delay product (EDP) as a metric for quantifying the energy-efficiency of GC-criticality-aware scheduler. EDP is the product of energy consumed by an application and its execution time, and thus emphasizes both energy-efficiency and performance. Figure 3.11 presents the reduction in energy-delay product across benchmarks and heterogeneous architectures, relative to *gc-on-small* which was designed for energy-efficiency [24]. Our scheduler beats *gc-on-small* by a significant margin—20% on average—for the GC-critical benchmarks on the 3B1S architecture. The key insight here is that when applications will be stalled due to slow collection, giving GC time on the big core can be more energy-efficient than always leaving it on small cores. GC-uncritical benchmarks, on the other hand, are affected neutrally in terms of energy efficiency, except for antlr which sees a slight increase in EDP, for the reasons explained with Figure 3.9.

3.7.3 Scaling Small Core Frequency

While the previously-presented results are more conservative due to the big and small cores running at the same frequency, we now explore scaling down the frequency of the small core. Figure 3.12 presents the results of our GC-criticality-aware scheduler as we change the small core's frequency from the default of 2.66 GHz to 1.66 GHz

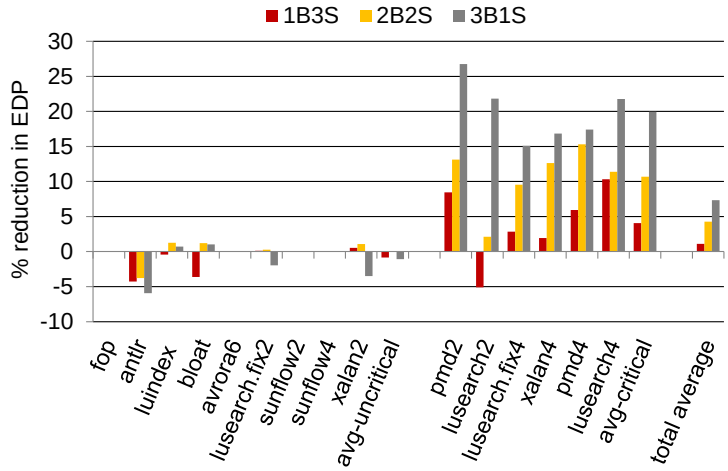


Figure 3.11: Percentage reduction in energy-delay product. *By improving performance, GC-criticality-aware scheduling also improves energy efficiency over keeping GC threads on small cores.*

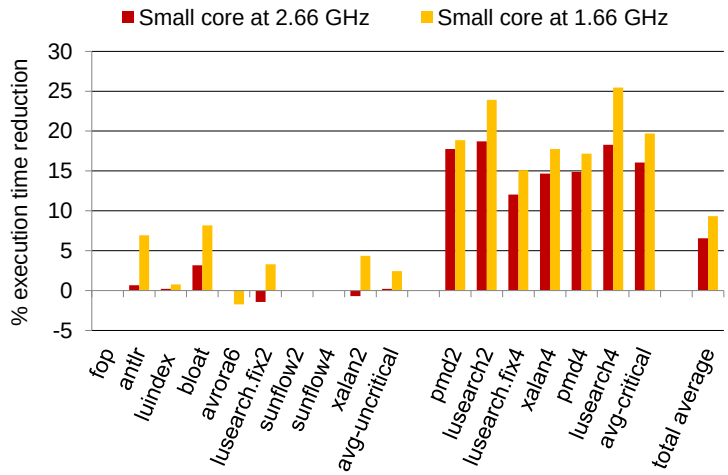


Figure 3.12: Percentage execution time reduction for 3B1S when scaling down the frequency of the small core. *GC-criticality-aware scheduling improves performance more when small cores run at a lower frequency than big cores.*

with a 3B1S architecture. When the small core is run at a lower frequency, even some of the benchmarks categorized as GC-uncritical exhibit GC criticality in their runs. This happens because application threads run at a higher frequency compared to GC threads, which default to be on the small core and thus cannot keep up with allocation demand. With the 1.66 GHz small core, antlr, bloat, lusearch-fix2, and xalan2 see performance improvements that they did not see with 2.66 GHz. Furthermore, all six GC-critical applications see even larger execution time reductions with the small core's scaled-down frequency, leading to the GC-critical average of 20% better performance.

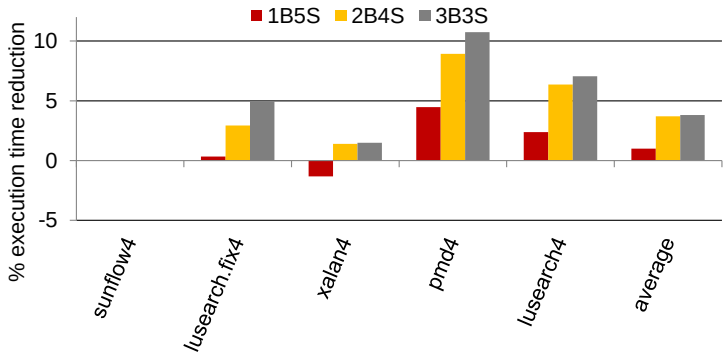


Figure 3.13: Percentage execution time reduction of our adaptive scheduler for various heterogeneous architectures with six total cores and four-threaded applications. *GC criticality still exists when thread count equals core count and GC-criticality-aware scheduling still improves performance.*

3.7.4 Larger Core Counts

Figure 3.13 presents performance results for heterogeneous architectures with six total cores, varying the number of big cores: 1B5S, 2B4S, 3B3S. Our scheduler uses its default parameters of $T_s = 100$ and $I_{max} = 8$. We present results for only four-threaded applications as then the number of threads and cores are equal, modeling a non-over-subscribed system. With equal numbers of threads and cores, all threads have the opportunity to progress, and thus we see less GC criticality in these configurations. However, while gains are modest, particularly for 1B5S, GC criticality does still exist for these four-threaded applications (besides sunflow4). Particularly with more big cores to divide between application and GC threads, GC-criticality-aware scheduling achieves performance improvements.

3.7.5 Heap Size Sensitivity Study

We now show experiments varying the heap size. Figure 3.14 plots execution time reductions for our adaptive scheduler with a smaller and larger heap: 0.75× and 1.5× the default (in Table 3.3) used in other experiments, for the 3B1S architecture. Our algorithm performs well across heap sizes. As when the small core's frequency is scaled down, we see that benchmarks previously labeled GC-uncritical, such as luindex, bloat, lusearch-fix2, and xalan2, exhibit GC criticality when the heap size is reduced, and have significant performance benefits from using our dynamic scheduler: up to 18%. For three GC-critical applications, we also see higher improvements with a smaller heap size, indicating that our scheduler will be more beneficial with a more constrained memory system. Naturally, when the heap grows, the application has more space into which to allocate, and thus does not encounter GC STW scan phases as much. GC-critical benchmarks, however, still realize performance improvements at the larger heap size: on average 5%. We also performed experiments using a larger trigger value (32 MB compared to our default 8 MB), which initiates concurrent collection cycles less frequently. The results show that as expected, similar to using larger

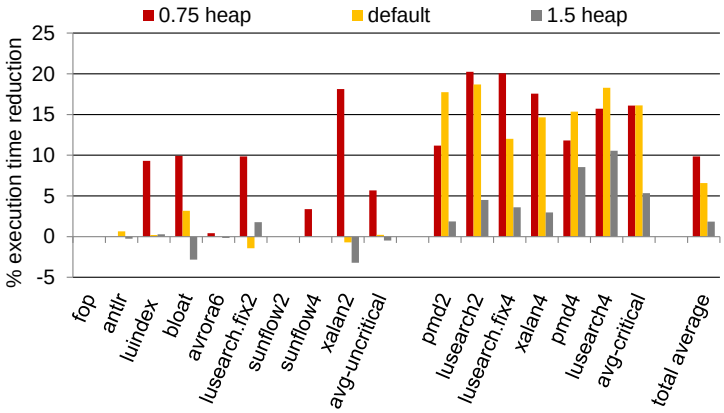


Figure 3.14: Percentage execution time reduction of our adaptive scheduler with varying heap sizes (ratios of those in Table 3.3) on 3*B*1*S*. *GC-criticality-aware scheduling performs well across heap sizes.*

heap sizes, GC criticality is reduced because collection runs concurrently with the application less. However, GC-critical benchmarks still experience scan pauses, and our GC-criticality-aware scheduler still realizes significant performance improvements.

3.8 GC Criticality in OpenJDK

In this section, we demonstrate that some applications still exhibit GC criticality in a different environment, namely, when run on top of OpenJDK 6 HotSpot JVM using its Concurrent Mark Sweep (CMS) collector. We perform experiments on a real machine using frequency scaling. Unlike Jikes' concurrent collector, which is not generational, OpenJDK's CMS collector is generational. We perform experiments on an eight-core machine with two Intel Xeon X5570 processors. We modify OpenJDK to pin threads that perform the GC-related tasks to a separate socket (Socket-GC). All other threads, including other service threads, run on their own socket (Socket-App). These threads are easily identified in OpenJDK by noting their ThreadType attribute (pgc-thread and cgc-thread). We run multithreaded CMS with two concurrent collection threads and two stop-the-world threads. We use the same heap size as before. The frequency of each socket can be scaled between 1.6 GHz and 3.0 GHz. We fix the frequency of the socket running the application threads and the JVM service threads (other than the GC threads) to 3.0 GHz (Socket-App). We run benchmarks from the DaCapo suite twice, first running the Socket-GC at 3.0 GHz, and then at 1.6 GHz. Figure 3.15 plots the percentage increase in execution time when GC threads run at 1.6 GHz versus 3.0 GHz. Some benchmarks do not suffer from running the GC socket at 1.6 GHz, which is advantageous in terms of energy-efficiency. However, seven benchmarks observe a more than 4% increase in execution time, and up to 8%. All benchmarks (to the right) that we identified as GC-critical when running with Jikes' concurrent collector are also GC-critical with OpenJDK's concurrent collector. Our GC-criticality-aware scheduler aims to mitigate this performance loss by preemptively boosting concurrent GC threads to have more big core machine resources when they are observed to be

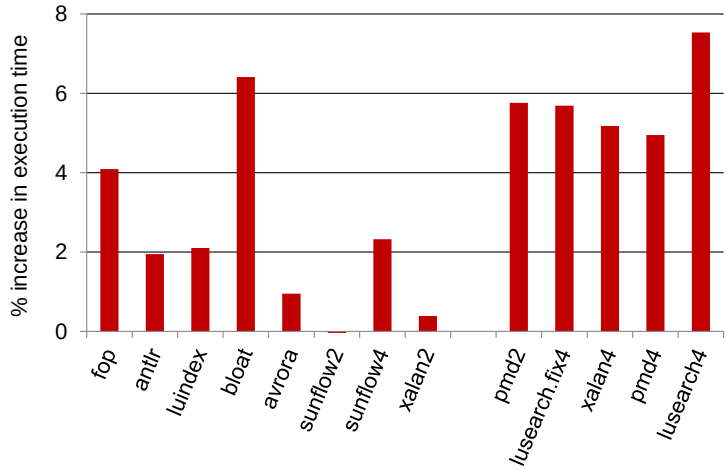


Figure 3.15: Total execution time increase for OpenJDK when the GC threads, isolated on a separate socket, are run at 1.6 GHz versus 3.0 GHz. Results are with OpenJDK's concurrent generational mark-sweep collector.

critical.

3.9 Summary and Interpretation

In this chapter, we study how to schedule managed language applications, and concurrent garbage collection in particular, on single-ISA heterogeneous multicores. We demonstrate, contrary to prior work, that concurrent garbage collection can significantly benefit (up to 18%) from out-of-order versus in-order execution. Moreover, we find that applications exhibit GC criticality when the application allocation rate exceeds the collection rate; in this case, it is then beneficial to take big core cycles from the application to give to concurrent GC threads so that they can collect the heap faster, avoiding costly stop-the-world pauses that make GC critical. These results motivate our novel adaptive scheduling algorithm that dynamically sets the GC's priority for getting big core cycles based on GC criticality signals from the managed runtime. GC-criticality-aware scheduling improves performance by 2.9%, 7.8%, and 16% and energy-delay product (EDP) by 3.5%, 10.7% and 20% for a set of GC-critical benchmarks when using one, two, or three big cores with four total cores, respectively; while being performance-neutral for GC-uncritical applications.

Our scheduling algorithm is reactive, adapting to benchmark phase behavior, dynamically changing GC's priority on the big core(s) while the benchmark runs. For phases and executions where GC is uncritical, our algorithm keeps GC threads on the small core(s), achieving similar energy efficiency and performance as when using a *gc-on-small* scheduler. For GC-critical phases, our algorithm boosts the GC threads' priority on the big core(s), beyond *gc-fair* if needed, balancing cycles between application and GC threads.

GC-criticality-aware scheduling is robust across core counts, big to small core

ratios, heap sizes, and clock frequency settings. Our results indicate that the importance of GC-criticality-aware scheduling increases as GC becomes more critical. Furthermore, GC can become critical for a number of reasons such as with applications running with more threads, on heterogeneous multicores with more big cores, with a smaller heap size, and/or with small cores running at a lower frequency. For popular managed applications, making schedulers aware of GC criticality leads to dynamically-optimized application performance and energy on future heterogeneous architectures.

The application and concurrent collection threads manifest the widely used producer-consumer pattern in concurrent programming. One class of threads produces information (mutator or application) and another class of threads consumes the information (concurrent collection). Producers and consumers communicate via shared queues. The insights in this research generalize to applications that use the producer-consumer pattern. Slow producer threads can eventually stall the consumer threads and vice-versa. Communicating the relative progress of producers and consumers to the OS scheduler can lead to better scheduling decisions on a hetrogeneous multicore.

Chapter 4

DVFS Performance Prediction with DEP+BURST

4.1 Introduction

In modern times, improving the energy-efficiency of computer systems is of prime importance. One way to manage the processor's power and energy consumption is using Dynamic Voltage and Frequency Scaling (DVFS). DVFS allows one to simultaneously change a processor's voltage and frequency. To effectively utilize DVFS, we need the ability to predict its performance impact on applications at run-time. Accurate DVFS performance prediction enables different opportunities for reducing the energy consumed by our applications. In particular, two possibilities include reducing the energy consumption while honoring a user-specified performance constraint, or running applications at the frequency that minimizes total energy consumption.

During the last decade, significant progress has been made in understanding and predicting the performance impact of DVFS for native sequential applications written in C and C++, see for example [44, 106, 70, 92, 127, 31]. However, prior work lacks a DVFS predictor for multithreaded applications, especially those written in managed languages, such as Java.

Existing DVFS predictors for sequential applications view a processor core as either executing instructions or waiting for memory accesses to return. The time spent executing instructions scales with frequency, whereas the time spent waiting for memory does not. Although this view suffices for sequential applications, it is not sufficient for multithreaded applications. For one, synchronization activity in multithreaded applications leads to inter-thread dependencies. Consequently, speeding up or slowing down one thread using DVFS impacts the execution of dependent threads, leading to complex interactions which affect overall application performance. A DVFS predictor for multithreaded applications therefore needs to take into account synchronization when predicting the total execution time at the target frequency.

Managed applications, which run on top of a virtual machine, exhibit even more

inter-thread dependencies than native applications. Service threads, such as those that perform garbage collection and just-in-time compilation, run alongside the application threads [24, 104]. Application and service threads need to synchronize from time to time, leading to increased synchronization activity, which further complicates DVFS performance prediction.

An additional complication is that managed applications issue bursts of store operations. These occur for two reasons: due to garbage collection activities that move memory around, and due to the zero-initialization upon fresh allocation that many managed languages, such as Java, require to provide memory safety. Current predictors ignore store operations assuming they are not on the critical path. We find that ignoring store operations leads to incorrect DVFS performance prediction for managed applications.

In this work, we propose DEP+BURST, a novel DVFS performance predictor for managed multithreaded applications. DEP+BURST consists of two key components, DEP and BURST. DEP decomposes the execution time of a multithreaded application into epochs based on the synchronization activity of both the application and service threads. We predict the duration of epochs at a different frequency, and aggregate the predicted epochs while taking into account inter-thread dependencies to predict the total execution time at the target frequency. A crucial component of DEP is its ability to predict critical threads across epochs. BURST identifies store operations that are on the application's critical path, and predicts their impact on performance across different frequency settings. Based on a run at the baseline frequency of 1 GHz, DEP+BURST achieves an average absolute error of 6% when predicting performance at a 4 GHz target frequency, for a set of multithreaded Java applications from the DaCapo suite [14]. DEP+BURST's error is a significant decrease from the 27% error achieved by M+CRIT, a multithreaded extension of the state-of-the-art CRIT [92] performance predictor.

We integrate DEP+BURST into an energy management framework for managed applications. We first use the energy manager to reduce the processor's energy consumption by tolerating a slowdown in performance compared to running at the highest frequency. Our energy manager is able to dynamically select DVFS settings that result in energy savings in return for a slowdown of the application close to a user's expectation. On average, for a user-specified slowdown threshold of 5% and 10%, our energy management framework reduces energy consumption by 13% and 19%, respectively, for a set of memory-intensive applications. In a second use case, the energy manager optimizes for total system energy consumption, including that of the processor plus DRAM. On average, our energy manager reduces total energy consumption by 15.6% through dynamically responding to application's phase behavior to pick a frequency per time quantum that lowers total system energy. For each use case, we compare against an oracle scheme that explores all possible frequency settings, and for a number of benchmarks, we outperform this scheme by exploiting dynamic phase behavior.

This chapter makes the following contributions:

1. We identify that inter-thread dependencies and store bursts need to be taken into account to have an accurate performance predictor for multithreaded managed

applications.

2. We introduce a performance predictor, DEP+BURST, that significantly lowers the error of accurately predicting performance when scaling the frequency.
3. We perform two case studies with an energy manager that 1) targets reducing the processor's energy consumption by slowing down a program not more than a user-specified slowdown threshold, and 2) optimizes total system energy, taking memory's energy consumption into account.
4. We perform experiments exploring the scalability of DEP+BURST, the ramifications of having a coarser frequency step setting, and the execution time overhead of running the proposed DVFS predictor.

Having an accurate performance predictor for DVFS is crucial to maintaining good performance, especially for multithreaded managed applications, while reducing energy consumption.

4.2 Related Work

In this section, we discuss three areas of related work. With this context, the next section discusses background on existing DVFS performance predictors.

4.2.1 DVFS Performance and Power Prediction

Performance and power prediction is either done using analytical models or using regression models. Section 4.3 discusses previously proposed analytical DVFS performance predictors in great detail [45, 106, 70, 92, 127, 113, 31]. These papers introduce new hardware performance counters specifically for the purpose of predicting the performance impact of DVFS. Su et al. [116] have recently shown how to implement the Leading Loads DVFS predictor on real AMD CPUs. In contrast, other works propose regression models that are built using offline training to predict the power and performance impact of frequency and architectural changes [33, 80, 117]. To build a regression model, these works leverage existing hardware performance counters to measure various microarchitectural events.

Deng el al. [36] propose an algorithm to manage DVFS for both the processor and the memory while honoring a user-specified slowdown threshold. However, this and many other works on DVFS power management do not consider multithreaded applications.

In this work, we investigate predicting the performance impact of chip-wide DVFS settings. Prior work investigates the potential of per-core DVFS to manage the energy consumption of multithreaded applications [71, 55]. However, we leave this for future work.

4.2.2 Scheduling Multithreaded Applications

Recently, there is increased interest in scheduling multithreaded applications on multicore hardware to optimize performance and energy. The main focus to date is in identifying and accelerating bottlenecks in multithreaded code, such as serial sections, critical sections, and lagging threads [10, 118, 66, 65, 39]. Accelerated Critical Sections (ACS) is a technique that leverages support from the ISA, compiler, and the large cores on a single-ISA heterogeneous multicore to accelerate critical sections [118]. Unlike accelerating only critical sections, Bottleneck Identification and Scheduling (BIS) also targets other bottlenecks that occur during the execution of a multithreaded application such as serial sections, lagging threads, and slow pipeline stages [65]. The above works use ISA and compiler support to delimit bottlenecks in software, and use this information during execution to accelerate bottlenecks. On the other hand, Criticality Stacks, proposed by Du Bois et al. [39], identify critical threads in multithreaded applications by monitoring synchronization behavior.

Finally, when running multithreaded applications on heterogeneous multicore processors, an important goal is to prevent one or more threads from lagging behind other threads. To this end, Van Craeynest et al. [122] propose a fair scheduler for multithreaded applications that provides a fair share of the big, out-of-order cores to each thread in a heterogeneous multicore processor. Akram et al. [3] propose a GC-criticality-aware scheduler for managed language applications on heterogeneous multicores.

4.2.3 Energy Management

Prior work has proposed frameworks to manage power, energy and thermals through DVFS, hardware adaptation and heterogeneity for multithreaded applications [89, 99, 39]. Although managed code is now ubiquitous and used in many application domains and run on a variety of hardware substrates, relatively few works have looked into the energy management of managed applications. Sartor et al. [107] explored the potential of DVFS for managed applications, teasing apart the performance impact of scaling the frequency of application and service threads in isolation. However, their work does not propose an analytical model to quantify the performance impact. Other works that shed light on different aspects of managed applications relating to energy consumption include [108, 24, 43].

4.3 Background and Motivation

Chapter 2 provides background on managed languages and garbage collection. In this section, we first provide background on the state-of-the-art predictor for sequential applications. We then describe the challenges introduced by multithreading and managed languages. Finally, we discuss naive extensions to the state-of-the-art predictor to predict the performance of multithreaded managed applications.

4.3.1 DVFS Performance Predictors for Sequential Applications

The impact of changing the frequency on application performance is easily understood by dividing execution time into 'scaling' and 'non-scaling' components. The scaling component scales in proportion to frequency; the non-scaling component remains constant when changing frequency. This simple division of execution time into scaling and non-scaling components works because changing the processor's frequency does not alter DRAM service time, whereas an increase or decrease in processor frequency has a proportional impact on the rate at which instructions execute in the core pipeline. The key challenge for accurately predicting the performance impact of DVFS is due to the out-of-order nature of modern processor pipelines in which memory requests are resolved while executing and retiring other instructions. Three DVFS performance predictors have been proposed over the past few years for sequential applications, with progressively improved accuracy. We now briefly discuss these three predictors.

Stall Time. The simplest, and least accurate, of the three models is the *stall time* model [44, 70], which estimates the non-scaling component by measuring the time the pipeline is unable to commit instructions. The non-scaling component is underestimated because it does not account for the fact that instructions may commit underneath a memory access. The simplicity of this model implies that it is easy to deploy on real hardware using existing hardware performance counters.

Leading Loads. Proposed by three different groups around the same time [44, 70, 106], the *leading loads* model computes the non-scaling component by accounting for the full latency of the leading load miss in a burst of load misses. Modern out-of-order pipelines are able to exploit memory-level parallelism and handle independent long-latency load misses simultaneously. The leading loads model assumes that each long-latency load miss incurs roughly the same latency, and hence, for a cluster of long-latency load misses, the miss latency of the leading load is a good approximation for the non-scaling component. Recent work shows that the leading loads model can be deployed on real hardware by using performance counters available on modern processors [117].

CRIT. A fundamental limitation of the leading loads model is that it does not take into account that long-latency load misses may incur variable latency, for a variety of reasons, including memory scheduling, bank conflicts, open page policy, etc. This leads to prediction inaccuracy for the leading loads model, which is overcome by CRIT, the state-of-the-art DVFS predictor proposed by Miftakhutdinov et al. [92]. CRIT identifies the critical path through a cluster of long-latency load misses to model a realistic, variable-latency memory system. CRIT includes an algorithm to identify dependent long-latency load misses and uses their accumulated latency as an approximation for the non-scaling component. We will use CRIT as our DVFS performance predictor for an individual thread. Note that as of today, no implementation of CRIT exists on real hardware.

4.3.2 Challenges in DVFS Performance Prediction for Managed Multithreaded Applications

There are three major challenges for predicting the performance impact of DVFS for multithreaded managed applications.

Inter-thread dependencies due to multithreading. To protect shared variables, different threads of a multithreaded application use synchronization primitives. Common examples of synchronization include critical sections and barriers. Synchronization leads to inter-thread dependencies. No thread is allowed to continue past the barrier until all threads reach the barrier. The slowest thread determines the barrier execution time at the target frequency. With a critical section, the progress of a thread waiting for a lock will depend on how fast the thread currently holding the lock is progressing at the target frequency. Scaling the frequency of one thread in a multithreaded application impacts the execution of other dependent threads, affecting overall performance in a non-trivial way.

Interaction between application and service threads. A managed language execution engine, such as the Java Virtual Machine (JVM), consists of application threads and service threads. The most important service threads include garbage collection and just-in-time compilation. Application and service threads interact with each other. For instance, a stop-the-world garbage collector suspends the application for a short duration to traverse a region of memory called the heap, and reclaim memory being used by objects that are no longer referenced. To estimate the total execution time at a different frequency, a DVFS predictor thus needs to take the interaction between application threads and service threads into account.

Store bursts. To provide memory safety, the Java programming language requires that a region of memory is zero-initialized upon fresh allocation. The process of zero-initialization leads to a burst of store operations that fill up the processor's pipeline. Another source of store bursts is the copying of objects during garbage collection. Ignoring store operations completely, as prior DVFS predictors do, leads to incorrect predictions for managed language workloads.

4.3.3 Straightforward Extensions of Prior Work

Before describing our new predictor in the next section, we first present two straightforward extensions of prior work to deal with multiple threads and, in the second case, service threads. We will quantitatively compare DEP+BURST against these naive extensions in the results section, and detail why these models are insufficient.

M+CRIT. We call the first predictor M+CRIT (short for multithreaded CRIT), which is generally applicable to any multithreaded application. M+CRIT uses the intuition that the execution time of a multithreaded application is determined by the critical (slowest) thread. We first use CRIT to identify each thread's scaling and non-scaling components at the base frequency. We then predict each thread's execution time at the target frequency. The thread with the longest predicted execution time is the critical thread. The execution time of the critical thread is also the total execution time of the application at the target frequency.

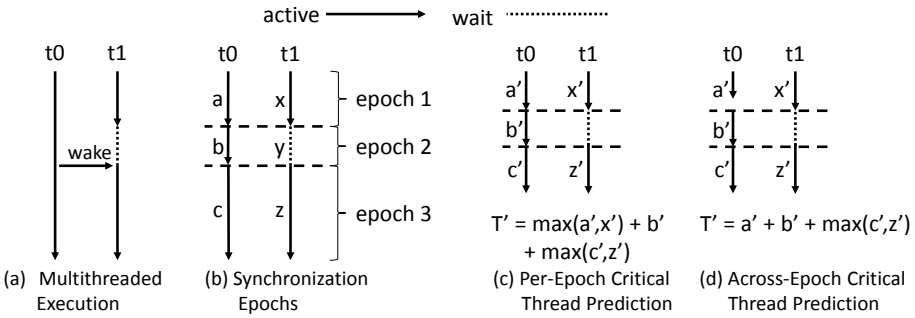


Figure 4.1: Showing how DEP breaks up a multithreaded application (a) into synchronization epochs (b) while running at the base frequency. DEP then estimates per-thread epoch durations at the target frequency, calculates the critical thread per epoch (c), and accounts for changes in the critical thread across epochs (d).

COOP. We term the second predictor COOP (short for cooperative), which is specific to Java applications. A typical Java application with a stop-the-world garbage collector goes through an 'application' phase, followed by a 'collector' phase. COOP intercepts the communication between the application and collector threads using signals from the JVM. Using these signals, COOP is able to distinguish application and collector phases. Once these individual phases are identified, COOP then uses M+CRIT to predict the execution time of the individual phases and aggregates the predictions to obtain a prediction for the total execution time.

4.4 The DEP+BURST Model

We now discuss our new DVFS performance predictor for managed multithreaded applications in detail.

4.4.1 Overview

Our proposed DVFS predictor estimates the performance of a managed multi-threaded application in two steps. In the first step, the predictor decomposes execution time into epochs based on synchronization activity in the application to account for inter-thread dependencies and the interaction between the application and service threads. In the second step, the predictor estimates the execution time of each active thread at a target frequency, taking into account which thread is critical and adjusting for dependencies with other epochs. Our model, which we call DEP, estimates the epoch execution time at the target frequency, and aggregates epochs to predict the total application execution time. To additionally take into account store bursts, we modify the second step to adjust the calculation of the scaling and non-scaling portions per thread within an epoch. When accounting for store bursts, we call our full model DEP+BURST. In the following sections, we first describe DEP, and then BURST.

4.4.2 Identifying Synchronization Epochs

First, we describe how DEP decomposes execution time into *synchronization epochs*. A synchronization epoch consists of a variable number of threads running in parallel. Two events mark the beginning of a new synchronization epoch: a thread is scheduled out by the OS and put to sleep, or a sleeping (or newly spawned) thread is scheduled onto a core. In multithreaded applications, threads typically go to sleep when access to a critical section is not available, or sleep while waiting at a barrier for other threads to join.

We identify synchronization epochs by intercepting the futex_wait and futex_wake system calls. Multithreading libraries such as pthreads use futexes, or fast kernel space mutexes [47] for handling locking. In the uncontended case, the application acquires the lock using atomic instructions without entering the kernel. Only in the contended case does the application invoke the kernel spin locks using the futex interface. Intercepting futex calls incurs limited overhead (less than 1%) [40].

To understand why futex-based decomposition is necessary, consider the example of a multithreaded execution in Figure 4.1(a). Two threads t0 and t1 from the same application are running in parallel. When t1 attempts to enter a critical section, t0 is already executing the critical section, which leads to t1 being scheduled out and made to wait for t0 to finish executing the critical section. When t0 is done executing the critical section, t1 is woken up.

An intuitive way to estimate the execution time of the example in Figure 4.1(a) is to first identify the non-scaling component of t0 and t1 when running at the base frequency, and subtract those from the total execution time to obtain the scaling components. This is what M+CRIT does. Using these per-thread components, it is straightforward to estimate the execution time of individual threads at the target frequency (see Section 4.3). Then the estimated execution time of the slowest thread serves as an estimate of total execution time. However, this leads to an incorrect estimation of execution time. The non-scaling component of execution time is actively accumulated in a counter only during the time a thread is active. During the time t1 is waiting, its non-scaling component depends on the activity taking place on the core where the thread holding the lock is running (t0). In the simple approach, the time t1 is waiting gets incorrectly attributed to the scaling component. Accurately estimating the execution time requires taking the dependency between t0 and t1 into account.

Figure 4.1(b) shows how our predictor decomposes the execution shown in Figure 4.1(a) into three epochs. a and x represent the duration of the first epoch, for threads t0 and t1, respectively. While these values are equal at the base frequency, we label them differently per thread because these values could be different when estimating time at the target frequency. b represents the duration of time that t0 is active during the second epoch when running at the base frequency. Similarly, c and z represent the duration of the third epoch. By decomposing execution time into epochs, DEP is able to model the dependency between t0 and t1 by analyzing b and predicting the new duration of b at a different frequency, which affects when both threads begin the third epoch at the new frequency.

It should be noted that the synchronization incurred by service threads, namely

between garbage collection threads, *and* the coordination between application and garbage collection threads is also communicated through futex calls. Therefore, by breaking down execution into epochs, we not only model the inter-thread dependencies between the application threads, but also account for the extra interactions between managed language application and service threads.

4.4.3 Predicting Performance at a Target Frequency

We now discuss how DEP estimates the duration of an epoch at a target frequency. During an epoch, DEP uses CRIT to accumulate the non-scaling component of each active thread in a counter. At the end of an epoch, both the scaling and non-scaling components are known. This provides DEP with a prediction of the duration of each thread at the target frequency. This is shown in Figure 4.1(c) and Figure 4.1(d) where a', b' and c' represent the estimated duration of t0's first, second and third epoch, respectively, at the target frequency. Similarly, x' and z' is the estimated duration of t1's first and third epoch at the target frequency. The next goal is to predict the execution time of an epoch from these individual estimates of all the active threads.

Per-epoch Critical Thread Prediction (CTP). An intuitive approach is to take the duration of the thread that runs the longest in the epoch, i.e., the critical thread, as the duration of the epoch at the target frequency. This approach is shown in Figure 4.1(c). This approach is simple to implement and does not require any bookkeeping across epochs. This technique does model the dependency between threads t0 and t1 in our running example and predicts when the third epoch would begin for both threads in the target frequency. However, using per-epoch critical thread prediction does not result in an accurate estimate of total execution time.

Across-epoch Critical Thread Prediction (CTP). We add across-epoch critical thread prediction to our DEP model to make it more accurate. This is shown in Figure 4.1(d). In the figure, a' is estimated to be shorter than x'. But if x' is taken as the duration of the first epoch, this leads to an incorrect estimation of the duration of the three epochs i.e., x' + b' + max(c',z'). The correct duration is a' + b' + max(c',z'), because thread t0 would just continue running after a' time units. In effect, part of x' gets overlapped with b' at the target frequency. Therefore, during each epoch, we need to store extra state to be able to identify the identity and duration of the critical thread to take that into account across epochs. Following the current example, we store the delta, x' - a', in a separate counter at the end of the first epoch. We also speculatively estimate the total execution time at the end of first epoch to be x'. In the second epoch, we subtract the delta-counter from b'. This way, at the end of the second epoch, we correctly estimate the total execution time to be a' + b'.

Algorithm. Our algorithm for performing across-epoch critical thread prediction is shown in Algorithm 2. First, we introduce the terminology used in Algorithm 2. α_t represents the estimated duration of a thread t at the target frequency. δ_t is the difference between the estimated duration of thread t and the estimated duration of the critical thread; δ_t of the critical thread is zero. The first step in Algorithm 2 is to compute the estimated duration, α_t, of each thread using CRIT (line 2). Next, we calculate the 'effective' execution time (e_t) of each thread by subtracting δ_t from α_t

ALGORITHM 2: Algorithm for across-epoch CTP.

input : A synchronization epoch S (I time units)
input : Initial delta-counters (δ_t) of all threads
input : Identity of the stalled thread if any (stall_tid)
output Estimated duration (I' time units) of S at target frequency
:

```
1 for each active thread t in S do
2 |   α_t = computeEstimatedTimeUsingCRIT()
3 |   e_t = α_t - δ_t
4 end
5 I' = Largest e_t
6 for each active thread t in S do
7 |   δ_t = (I' - α_t) + δ_t
8 end
9 δ_stall−tid = 0
```

(line 3). The thread with the largest e_t is the critical thread, and the corresponding e_t is the duration of the epoch (I') (line 5). Note that δ_t is accumulated across epochs, with a term representing the difference between I' and α_t added during each epoch until the thread stalls (line 7). We reset δ_t of a stalling thread to zero (line 9).

4.4.4 Modeling Store Bursts

Store bursts occur more frequently in managed language workloads than in native applications. In Java in particular, store bursts originate from two main sources: (1) zero-initialization to provide memory safety, and (2) copying of objects during garbage collection. A DVFS model for Java applications should incorporate the impact of store bursts.

CRIT assumes that store instructions are not on the application's critical path. This is true for a few isolated store requests that miss in the L1 cache because the store queue provides modern processors with the ability to execute loads in the presence of outstanding stores (through load bypassing and store-to-load-forwarding). Furthermore, it contains committed stores until they are retired by the memory hierarchy, freeing up space in the ROB or active list. Normally, the work done underneath a store miss scales with frequency. However, a fully-occupied store queue stalls the processor pipeline. Store bursts fill up the store queue before eventually stalling the pipeline.

In typical out-of-order pipelines, an entry is allocated in the ROB and the store queue at the time the store instruction is issued. When a store commits from the head of the ROB, the entry is no longer maintained in the ROB, but the entry is maintained in the store queue until the outstanding request is finally retired. Commit stalls when the store queue is full and the next instruction to commit is a store.

To account for store bursts, we accumulate the amount of time the store queue is full in a counter when running at the base frequency. For each active thread during an epoch, we add the counter's contents to the non-scaling component measured by CRIT. When modeling the impact of store bursts, we add BURST next to the model

name. Thus, our proposed model that takes both inter-thread dependencies and store bursts into account is called DEP+BURST.

4.4.5 Implementation Details

Now, we discuss implementation issues when porting DEP+BURST to real hardware. First, the OS is the best place to identify synchronization epochs, for instance, as a kernel module. The OS is aware of thread creation, deletion, and other events regarding thread scheduling including the futex_wait and futex_wake system calls.

Multiple threads time-sharing a single core is a common practice to consolidate resources. In such a case, the OS periodically schedules out the currently executing thread out of the core, and schedules one of the waiting threads in. Whenever that happens, we start a new epoch. As a result, time-multiplexing cores among threads is seamlessly handled by DEP.

We require extra counters to implement DEP+BURST on real hardware. We use CRIT [92] within an epoch to divide a thread's execution into scaling and non-scaling portions, so our model requires the same bookkeeping information as CRIT.

Tracking store bursts requires simple additional logic in the store queue that generates a signal once all its entries are occupied. The performance counter hardware monitors this signal to account for the time the store queue is full.

To account for critical threads across epochs, we require one counter per thread. This counter can be maintained in software inside the kernel module that intercepts the futex calls.

4.5 Experimental Methodology

Before evaluating the accuracy of DEP+BURST, we first describe our experimental setup.

Simulator. We use Sniper [26] version 6.0, a parallel, high-speed and cycle-level x86 simulator for multicore systems; we use the most detailed cycle-level core model available. Sniper was further extended [108] to run a managed language runtime environment including dynamic compilation, and emulation of frequently used system calls.

Benchmarks. We use seven multithreaded Java benchmarks from the DaCapo suite [14] that we can get to work with Jikes RVM 3.1.2 [4]. We use five benchmarks from the DaCapo-9.12-bach benchmark suite (avrora, lusearch, pmd, sunflow, xalan). We also use an updated version of lusearch, called lusearch-fix (described in [129]), that eliminates needless allocation. Finally, we use an updated version of pmd, called pmd-scale (described in [40]) that eliminates the scaling bottleneck due to a large input file. All benchmarks we use in this work are multithreaded. The avrora benchmark uses a fixed number (six) of application threads, but has limited parallelism [40]. For the remaining multithreaded benchmarks, we evaluate using four application threads. Table 4.1 lists our benchmarks, a classification of whether they are memory or compute-

Table 4.1: Our benchmarks from the DaCapo suite, including a classification of their type, heap size, execution time and GC time at 1 GHz. M represents a memory-intensive benchmark, and C represents a compute-intensive benchmark.

Benchmark	Type [M/C]	Heap size [MB]	Execution time (ms)	GC time (ms)
xalan	M	108	1,400	270
pmd	M	98	1,345	230
pmd.scale	M	98	500	80
lusearch	M	68	2,600	285
lusearch.fix	C	68	1,249	42
avrora	C	98	1,782	5
sunflow	C	108	4,900	82

intensive, the heap size we use in our experiments (reflecting moderate, reasonable heap pressure [108]), and their running time when using Sniper with each core running at 1 GHz. We classify the benchmarks based on the intensity of garbage collection. An application that spends more than 10% of its execution time in garbage collection is considered a memory-intensive benchmark. lusearch.fix, avrora, and sunflow are compute-intensive, and the remaining five benchmarks are memory-intensive.

Processor architecture. We consider a quad-core processor configured after the Intel Haswell processor i7-4770K, see Table 5.2. Each core is a superscalar out-of-order core with private L1 and L2 caches, while sharing the L3 cache. We vary the cores' frequency between 1 and 4 GHz.

Power and energy modeling. We use McPAT version 1.0 [86] for modeling power consumed by the processor. For DVFS support, we use the Sniper/McPAT integration described in [54] while considering a 22 nm technology node. We use a frequency step setting of 125 MHz when dynamically adjusting the frequency to save energy (Section 4.7). We use the voltage and frequency settings for a 22 nm technology node, closely following Intel's i7-4770K (Haswell) [34]; see Table 5.2 for a subset of settings[1]. When reporting power numbers, we include both static and dynamic power. We model the DVFS transition latency as a fixed cost of 2 μs [57].

4.6 Model Evaluation

We now evaluate the accuracy of DEP+BURST. We first compare the accuracy of DEP against M+CRIT and COOP to understand the impact of taking inter-thread dependencies into account. We then evaluate DEP with and without BURST, teasing apart the contribution of taking store bursts into account.

[1]The remaining settings can be found using the linear relationship between core voltage (v) and frequency (f), v = 0.11 * f + 0.63 [34].

Table 4.2: Simulated system parameters.

Component	Parameters
Processor	4 cores, 1.0 GHz to 4.0 GHz
	4-issue, out-of-order, 128-entry ROB
	Outstanding loads/stores = 48/32
Cache hierarchy	L1-I/L1-D/L2, Shared L3 (1.5 GHz)
	Capacity: 32 KB / 32 KB / 256 KB / 4 MB
	Latency : 2 / 2 / 11 / 40 cycles
	Set-associativity: 4 / 8 / 16
	64 B lines, LRU replacement
Coherence protocol	MESI
DRAM	FR-FCFS, 12 GB/s, 45 ns latency
DVFS states	(1, 0.737); (1.5, 0.791); (2, 0.845);
(GHz,Vdd)	(2.5, 0.899); (3, 0.958); (3.5, 1.012);
	(4, 1.07)

4.6.1 Prediction Accuracy

Evaluating the accuracy of a DVFS performance predictor is done as follows. We run the application at both the baseline and target frequency. We predict the execution time at the target frequency based on the run at the baseline frequency, and we compare the predicted execution time against the measured execution time. We quantify prediction accuracy as the relative prediction error (estimated - actual) / actual. A negative error thus implies an underestimation of the execution time or a performance overestimation. The reverse applies for a positive error.

Evaluating a DVFS performance predictor requires choosing a baseline and target frequency. When used as part of an energy management framework — as we will explore in our case study — it is important that we are able to accurately predict performance both at higher and lower frequencies. We hence consider two scenarios: one in which we consider a low base frequency and predict performance at higher frequencies, and one in which we consider a high base frequency and predict performance at lower frequencies. Figure 4.2 quantifies the prediction error for all benchmarks (including the average absolute error) for three target frequencies when the base frequency is set at 1 GHz, i.e., predicting performance at a higher frequency than the baseline frequency. Figure 4.3 shows similar data for target frequencies smaller than the base frequency set to 4 GHz.

M+CRIT has the worst prediction error of all models. The average absolute error is 27% when predicting from 1 GHz to 4 GHz, and 70% when predicting from 4 GHz to 1 GHz. Clearly, not taking into account synchronization, inter-thread dependencies and store bursts leads to highly inaccurate DVFS performance prediction for managed multithreaded applications.

Taking into account the interaction of application and managed language service threads, as COOP does, slightly improves accuracy over M+CRIT. However, the

(a) Low-to-high prediction from a baseline at 1.0 GHz

Figure 4.2: Per-benchmark prediction errors at higher frequency from a baseline of 1 GHz for M+CRIT, COOP and DEP, both with and without BURST. *DEP+BURST outperforms all other predictors with an average absolute estimation error of below 10%.*

(a) High-to-low prediction from a baseline at 4.0 GHz

Figure 4.3: Per-benchmark prediction errors at lower frequency from a baseline of 4 GHz for M+CRIT, COOP and DEP, both with and without BURST. *DEP+BURST outperforms all other predictors with an average absolute estimation error of below 10%.*

prediction error is still significant with average absolute prediction errors for COOP of 22% and 63% for the base 1 and 4 GHz scenarios, respectively.

Taking all synchronization activity into account, as DEP does, further improves accuracy, with an average absolute error of 19% and 57% for the base 1 and 4 GHz scenarios, respectively. The conclusion from this result is that managed multithreaded applications require accurate modeling of inter-thread dependencies both through coarse-grained synchronization between application phases and garbage collection phases, as well as through fine-grained synchronization between application threads and between garbage collection threads. Unfortunately, although the prediction error is decreased compared to M+CRIT and COOP, DEP's error is still high.

Modeling store bursts brings the error down substantially, especially for the memory-intensive benchmarks. In fact, all three models, M+CRIT, COOP and DEP, benefit from BURST modeling. It is interesting to note that DEP benefits most from BURST, and COOP benefits more than M+CRIT. Because DEP more accurately identifies critical threads, adding the modeling of store bursts, which affect the critical thread, improves accuracy more substantially. Likewise, COOP identifies critical threads more accurately than M+CRIT, and hence benefits more from store burst modeling than M+CRIT.

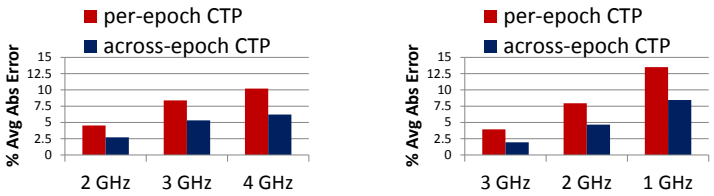
This leads to the overall conclusion that DEP+BURST is the most accurate DVFS performance predictor, with an average absolute error of 6% when predicting from 1 GHz to 4 GHz, and an average absolute error of 8% when predicting from 4 GHz to 1 GHz. This result shows that modeling both synchronization and inter-thread dependencies as well as store bursts is critical for DVFS performance prediction of managed multithreaded applications.

Prediction errors tend to increase for target frequencies that are 'further away' from the base frequency, due to accumulating errors, which is especially noticeable for memory-intensive applications. Further, when predicting the execution time in the high-to-low scenario, an error in incorrectly estimating the scaling component multiplies as the target frequency increases. This leads to increased inaccuracy in identifying the critical thread in an epoch. When predicting low-to-high, the scaling component is divided by a factor, making the error less prominent.

Explaining lusearch **and** avrora**.** From the results in Figure 4.2 and Figure 4.3, we note a higher estimation error for two benchmarks: avrora and lusearch. Our analysis indicates that each of the two benchmarks stresses a different component of DEP+BURST. avrora has the largest number of epochs among all of our benchmarks, pointing to a large number of inter-thread dependencies, thus stressing the DEP component. On the other hand, lusearch allocates the most memory. Its error is high because it is overly sensitive to the approximation we make to model store bursts.

4.6.2 Per-Epoch vs. Across-Epochs CTP

As argued in Section 4.4, it is important to accurately predict the critical thread at each point during the execution. We described two approaches to this problem, namely per-epoch critical thread prediction (CTP) and across-epoch CTP. We now quantify the importance of across-epoch CTP. Figure 4.4 reports the prediction error



(a) 1.0 GHz to 4.0 GHz (b) 4.0 GHz to 1.0 GHz

Figure 4.4: Comparing per-epoch versus across-epoch critical thread prediction. *Detecting critical threads across epochs leads to a more accurate predictor for multithreaded managed applications.*

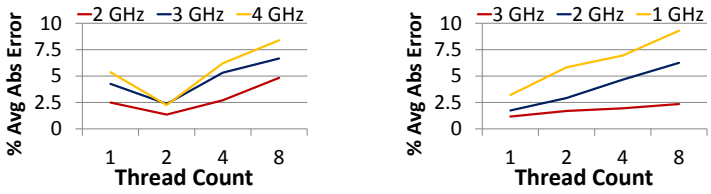
for DEP+BURST with across-epoch CTP versus per-epoch CTP. Across-epoch CTP brings down the average absolute error by a significant margin compared to per-epoch CTP: by 4% (from 10% to 6% average absolute error) at 4 GHz with a 1 GHz base frequency, and by 6% (from 14% to 8% average absolute error) at 1 GHz with a 4 GHz base frequency. This result confirms that being able to accurately predict the critical thread at all points during the execution time, and carry this dependence across epochs, is a key component of DEP+BURST.

4.6.3 Scalability

We have shown the accuracy of DEP+BURST with four application and two GC threads. We now experiment with different thread counts to explore our predictor's scalability. Increasing the number of application threads stresses the predictor in different ways. More application threads lead to more inter-thread dependencies. Increasing the thread count also increases the rate that store bursts are issued, because there is also more memory allocation. This is because thread-local storage increases the total amount of memory allocated when running benchmarks with more threads. Prior work also shows that the amount of work that GC performs increases with more application threads [40].

To understand the scaling behavior of our proposed model, we show the accuracy of DEP+BURST with 1, 2, 4 and 8 application threads. Because of the presence of service threads such as the garbage collector, managed environments are multithreaded even with one application thread. When running the benchmarks with one application thread, we use a single garbage collector thread. We use two garbage collector threads for experiments with more than one application thread. Prior work by Du Bois et al. [40] reports that Jikes' generational Immix garbage collector does not scale beyond two threads. We use a single core per application thread in our experiments. We set the last-level cache to have 1 MB/core and the memory bandwidth is set to 3 GB/s/core. Our modeled processors reflect many commercial designs on the market today.

Figure 4.5 shows the average absolute error of our predictor with different thread counts. The average error with one application thread is 5.4% when predicting from 1 GHz to the highest target frequency of 4 GHz, and 3.2% when predicting from 4 GHz to 1 GHz. Thus, our DEP+BURST predictor is also accurate for single-threaded



(a) 1.0 GHz to 4.0 GHz (b) 4.0 GHz to 1.0 GHz

Figure 4.5: The accuracy of DEP+BURST for different thread counts. *DEP+BURST's error increases only slightly as the number of threads is increased from one to eight.*

managed applications. As the thread count increases, the average absolute error goes up. However, for up to eight threads, the increase is not dramatic. When predicting from 1 GHz to higher frequencies, the average error with eight application threads is below 10% for any target frequency. The average error with more than one application thread is slightly higher when predicting from 4 GHz to 1 GHz (9.3% with eight threads). Note that the frequency range that we explore, 1 GHz to 4 GHz, represents a very wide frequency spectrum. Memory behavior is likely to change across such a wide frequency spectrum, making it harder to predict performance accurately, especially as the number of threads is increased.

4.7 Case Studies

Having described and evaluated the DEP+BURST DVFS performance predictor, we now use it in two case studies involving an energy manager. In the first case study, the energy manager leverages a performance predictor to reduce the processor's energy consumption without slowing down the application more than a user-specified threshold. In the second case study, the energy manager uses analytical performance and energy models to optimize the full system energy consumed by an application.

4.7.1 Case Study 1: Energy Minimization under Performance Constraints

It is well-known that it is possible to reduce the processor's energy consumption by lowering the frequency. The intuition is that lowering the frequency reduces power consumption, leading to a more energy-efficient execution. Lowering the frequency reduces energy consumption as long as the reduction in power consumption is not offset by an increase in execution time. This is typically the case for memory-intensive applications for which lowering the frequency incurs a small performance degradation. For compute-intensive applications on the other hand, the reduction in power consumption may be offset by an increase in execution time, leading to a (close to) net energy-neutral operation. In other words, different applications exhibit different sensitivities to scaling the processor's frequency. Moreover, compute- and memory-intensive phases may occur within a single application; this is especially the case for managed language workloads for which garbage collection is typically

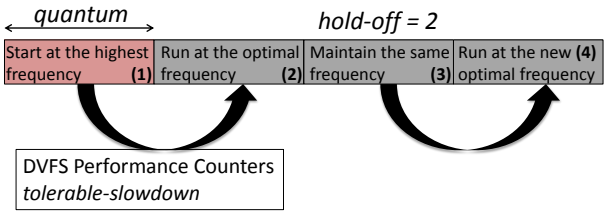


Figure 4.6: Example illustrating the energy manager using DVFS performance prediction. The input parameters of the energy manager are shown in italics.

memory-intensive [24, 104]. Hence, this calls for an energy management approach that dynamically determines when and to what extent to scale the frequency to minimize energy consumption while not exceeding a user-specified slack in performance.

Energy Manager

To demonstrate the importance of having an accurate DVFS performance predictor for multithreaded managed applications, we design an energy manager that minimizes energy consumption while guaranteeing performance within a user-specified threshold compared to running at the highest frequency. The high-level design is shown in Figure 4.6. The figure shows how the manager works for the first four intervals of the application. We always start the application at the highest frequency (4 GHz for our modeled processor). During this interval, the performance predictor reads the DVFS-related performance counters as described in Section 4.4. At the end of the first interval, the manager estimates performance at all of the DVFS states. The tolerable-slowdown is a user-specified parameter that the manager uses to identify all of the DVFS states that satisfy the performance constraint, i.e., performance is slowed down by no more than tolerable-slowdown, as a percentage compared to running at the highest frequency. Of all the states that satisfy the performance constraint, the manager then chooses the state with the minimum energy consumption (lowest frequency) for the next quantum. The hold-off parameter represents the number of intervals to wait before changing the frequency again. In the example shown in the figure, hold-off is set to two. Therefore, the third interval also runs at the same frequency as the second interval. In case the application has no phase behavior, using a large hold-off prevents needless profiling. The scheduling quantum is also an adjustable parameter, and is set to 5 ms in our experiments. We set the hold-off parameter to one unless otherwise specified.

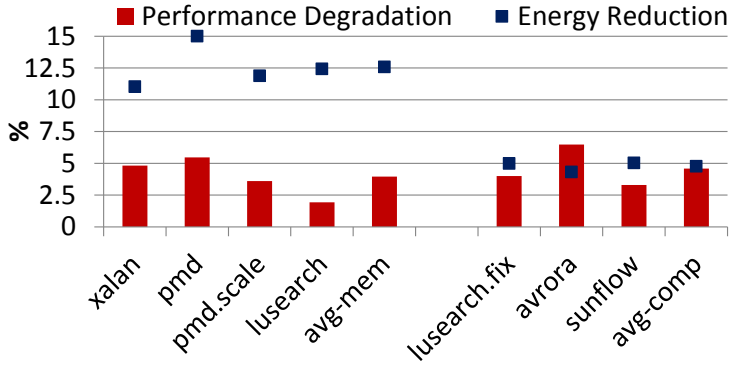
The key idea we use to guarantee that the application does not experience a slowdown more than the specified threshold is that, if each interval experiences a slowdown of x%, then the entire application experiences a slowdown of x% compared to always running the application at the highest frequency. To fulfill this requirement during each interval, we need to estimate the slowdown that the application experiences compared to running at the highest frequency, even when running at a slower frequency. We solve this problem in two steps. The energy manager first estimates the execution time at the highest frequency, before predicting execution time at the target frequency in the second step and its relative slowdown compared to running at the highest frequency. The manager finally chooses the minimum frequency setting that does not slow down

the next interval more than the user-specified threshold.

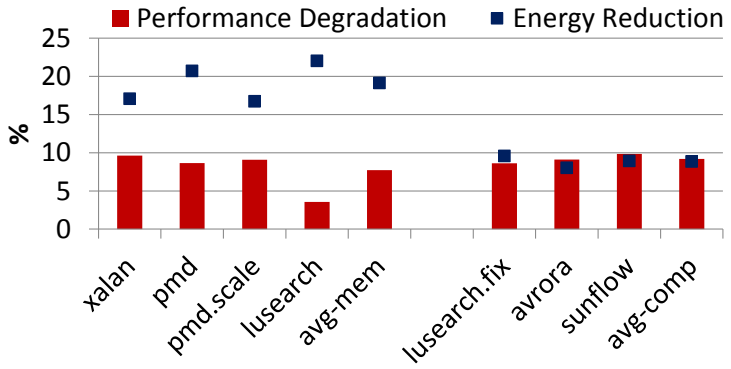
In the following sections, we explore the opportunity to reduce energy consumption with our proposed energy manager in detail.

Evaluation

Figure 4.7 reports the slowdown experienced by each benchmark and the corresponding reduction in energy consumption for user-specified slowdown thresholds of 5% and 10%. We observe substantial reductions in energy consumption for the memory-intensive benchmarks, by 13% on average (and up to 15%) for the 5% threshold, and by 19% on average (and up to 22%) for the 10% threshold. As expected, the reduction in energy consumption is not as significant for the compute-intensive workloads.



(a) Tolerable Slowdown = 5%



(b) Tolerable Slowdown = 10%

Figure 4.7: Per-benchmark reductions in energy consumption using DEP+BURST in our energy manager for a slowdown threshold of (a) 5% and (b) 10%. Memory-intensive benchmarks are to the left while compute-intensive are to the right. *Using the DEP+BURST predictor as part of our energy manager leads to a significant reduction in energy consumption for the memory-intensive benchmarks with only a slight performance degradation.*

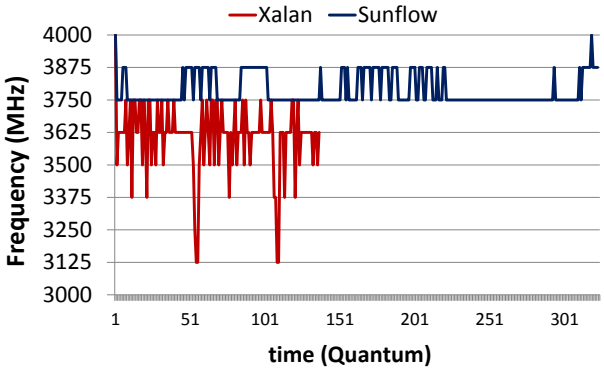


Figure 4.8: Per-quantum frequency settings chosen by the energy manager for xalan and sunflow for a slowdown threshold of 5%. *There is a larger variation in the processor's frequency during the execution of the memory-intensive benchmarks, compared to the compute-intensive benchmarks.*

It is interesting to note that the obtained performance is close to the user-specified performance target, i.e., the execution slowdown is around 5% and 10% for most benchmarks for the 5% and 10% thresholds, respectively. The benchmarks for which we observe an exception are avrora and lusearch, with a slight overshoot for avrora at the 5% threshold, and an undershoot for lusearch at both the 5% and 10% thresholds. The reason is the inaccuracy of the DVFS performance predictor: lusearch and avrora experience the largest prediction errors, as shown in Figure 4.2 and Figure 4.3. This result re-emphasizes the importance of accurate DVFS performance prediction for effectively managing energy consumption and performance when running managed multithreaded applications. Nevertheless, since lusearch stresses the memory subsystem the most, we observe a large reduction in its energy consumption despite slowing its execution less than the user-specified slowdown threshold.

Figure 4.8 shows the frequency settings chosen by our energy manager for xalan and sunflow for a slowdown threshold of 5%. The frequency settings chosen by our energy manger for the memory-intensive xalan cover a wider range compared to the compute-intensive sunflow. We observe similar trends in other benchmarks.

Comparison to static-optimal. To further analyze the robustness and importance of dynamically adjusting frequency, we compare our dynamic energy manager (using DEP+BURST) against the optimal frequency setting obtained statically. Static-optimal (Static-Opt) is determined by running the application multiple times offline, and selecting the optimal frequency that minimizes energy consumption across the entire run; because this static frequency is obtained while using the same input data set, we can consider the static-optimal frequency as an oracle setting. Note that Static-Opt is not a practical approach and is shown here for purposes of comparison only. Figure 4.9 compares the reduction in energy consumption by our dynamic energy manager to the reduction achieved by Static-Opt for a slowdown threshold of 10%. Our energy manager leads to larger reductions in energy consumed by all of the memory-intensive benchmarks with the exception of lusearch. For lusearch, DEP+BURST exhibits a larger error compared to the other benchmarks, which is the reason our energy manager

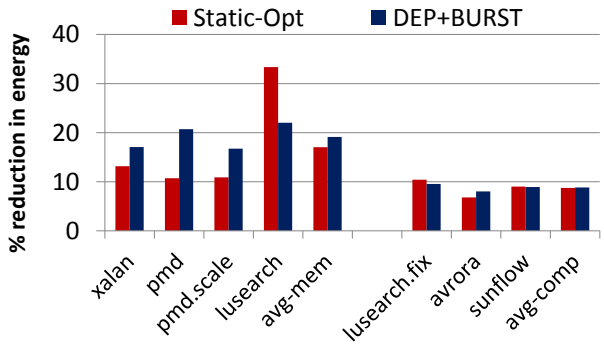


Figure 4.9: Reduction in energy consumption achieved by our energy manager compared to the static optimal (Static-Opt) for a slowdown threshold of 10%. *For six out of seven benchmarks, our energy manager reduces the energy consumption about the same as, or more than, that of Static-Opt.*

misses the full potential for reducing the energy consumption. The overall reduction is 2% on average and up to 10%. The reason why our energy manager outperforms Static-Opt for xalan, pmd and pmd.scale is because it is able to dynamically adjust the frequency in response to varying execution phase behavior, which Static-Opt, by definition, is unable to do. The reduction on average is on par with static-optimal for the compute-intensive applications.

Frequency Step Setting

The granularity at which you can change the frequency, or the step setting, is an important factor in meeting performance targets yet striving for energy efficiency. In the previous results, we assumed a frequency step setting of 125 MHz with a total of 25 DVFS settings between 1 GHz and 4 GHz. A coarser frequency step setting makes it difficult to meet the user-specified slowdown thresholds. For instance, assume that slowing down one phase of a benchmark by 5% requires a frequency setting of 3.8 GHz. If the machine only offers a frequency step setting of 500 MHz, our energy manager will run that benchmark's phase at 4 GHz, since running at 3.5 GHz slows down the phase by more than 5%. This leads to a missed opportunity to save energy.

We add extra accounting to our energy manager to keep track of these missed opportunities per phase so that in a later phase, the application can run at a lower frequency while still meeting the user-specified slowdown target over the entire run, thus reducing energy consumption. More specifically, during each profiling quantum, our energy manager stores the difference between the execution time of running at the ideal frequency that would get closest to the slowdown threshold and the execution time given the best-available frequency setting (less than the highest frequency and does not violate the slowdown threshold). In our example above, this would be the execution time difference when running the benchmark's phase at the desired 3.8 GHz versus the energy manager's chosen 4 GHz. We call this difference σ_{excess}, which is shown in Equation 4.1. $T_{desired}$ is the execution time running at some ideal frequency if we did have fine-grained step settings, and $T_{estimated}$ represents the estimated execution

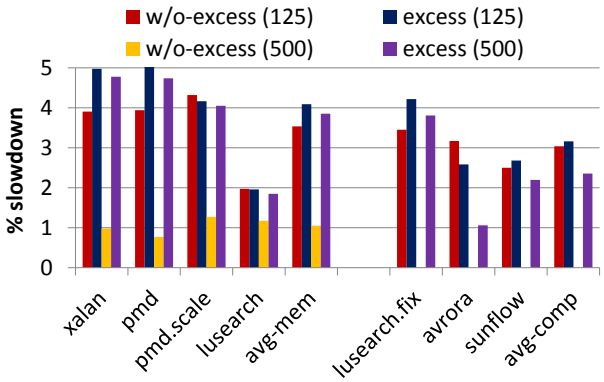


Figure 4.10: Per-benchmark slowdown for different frequency step settings with and without modeling excess-time. *Modeling excess-time results in an average slowdown close to the user's expectations, regardless of the available frequency step setting.*

time at best-available frequency setting. Note that $T_{desired}$ is just the time quantum's duration plus the user-specified slowdown. σ_{excess} is multiplied by the hold-off to account for there being no change in frequency for hold-off quantums.

$$\begin{aligned} T_{desired} &= quantum * (1 + tolerable\_slowdown) \\ \sigma_{\text{excess}} &= (T_{desired} - T_{estimated}) * hold\_off \end{aligned} \tag{4.1}$$

During a subsequent profiling quantum, the manager adds this previously calculated σ_{excess} to the execution time we want to achieve in this phase. For example, because we did not slow down the previous phase at all, even though we had a target of 5%, we can slow down the current phase by more than 5%. However, over the entire run, we expect to still meet the user's specified slowdown threshold, while reducing more energy consumption.

Figure 4.10 presents the per-benchmark slowdown with and without modeling σ_{excess} for two systems with a different number of DVFS states. One system provides a frequency step setting of 125 MHz (25 DVFS states), and the other system provides a frequency step setting of 500 MHz (7 DVFS states), which is more limiting. The slowdown threshold is set to 5%; hold-off is one; and the quantum length is 5 ms. With a frequency step setting of 125 MHz, the slowdown is 3.7% on average both with and without modeling σ_{excess}. However, with a 500 MHz frequency step setting, the average slowdown without modeling σ_{excess} (w/o-excess-500) is 0.7%, which is much lower than the 5% target. For several benchmarks, a 5% slowdown in execution time compared to running at 4 GHz is achieved by running at a frequency somewhere between 3.5 GHz and 4 GHz. However, running at 3.5 GHz is likely to slow down the execution more than 5% for these benchmarks. Therefore, the energy manager runs these benchmarks at the highest frequency during most of the execution. The average slowdown increases to 3.2% when modeling σ_{excess} (excess-500). In fact, all except one benchmark (avrora) experience a slowdown similar to excess-125.

Figure 4.11 shows the reduction in energy consumption with and without modeling σ_{excess}. For both the compute-intensive and the memory-intensive benchmarks, the

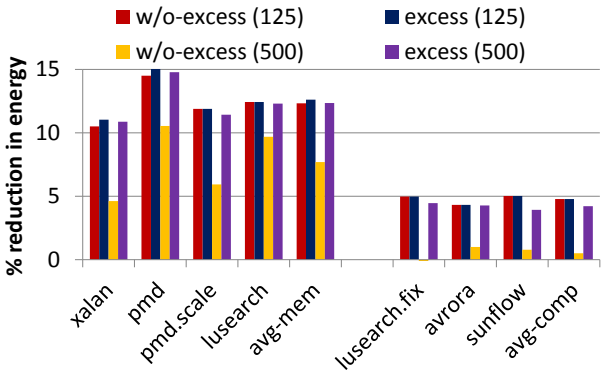


Figure 4.11: Per-benchmark reduction in energy consumption for different frequency step settings with and without modeling excess-time. *Modeling excess-time results in an average energy reduction for both the compute-intensive and the memory-intensive benchmarks regardless of the frequency step setting.*

energy reduction of w/o-excess-125 and excess-125 is almost the same. However, w/o-excess-500 achieves only 7.7% reduction in energy consumption for the memory-intensive benchmarks compared to 12.5% provided by excess-500. For the memory-intensive benchmarks, modeling σ_{excess} helps get closer to the user-specified slowdown threshold, and thus achieves a higher reduction in energy consumption. For the compute-intensive benchmarks, w/o-excess-500 barely provides any reduction in energy consumption at all. We conclude that modeling σ_{excess} is especially important with coarser frequency step settings, and that it makes our energy manager robust to whatever DVFS granularity is provided by the processor.

Varying Hold-off and Length of Profiling Quantum

In all previous experiments, we use a hold-off of one and a quantum length of 5 ms. In this section, we vary these parameters to see if it is possible to meet the user's execution time requirement while running the model less often. A large hold-off would translate to the energy manager running the model less often. Similarly, if the quantum is small, then the overhead of running the model is small.

In Figure 4.12, we vary the hold-off and keep the quantum length fixed at 5 ms. We show results with a hold-off of 1, 5, and 10. The tolerable slowdown is set to 5%. Using a large hold-off results in a slowdown further away from the user-specified threshold for three benchmarks, namely xalan, pmd and pmd.scale. Others either show no trend when increasing the hold-off or a slowdown slightly closer to the user's expectations. The average slowdown with a hold-off of 5, which runs the model less often, is 3.8%, compared to an average slowdown of 3.7% with a hold-off of one.

We show the impact of varying the quantum length on each benchmark's slowdown in Figure 4.13. We show results for five different quantum lengths, each a multiple of five. Because we are investigating the sensitivity to quantum length, we keep the hold-off at one. As before, the slowdown threshold is 5%. Each benchmark is affected differently, and we observe no general trend. For instance, avrora and pmd generally

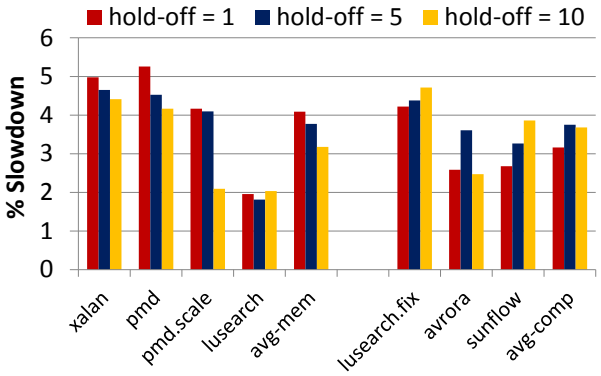


Figure 4.12: Per-benchmark slowdown for different values of hold-off. *On average, a hold-off of 5 is a good compromise between running the model less often, and being close to the user-specified slowdown threshold.*

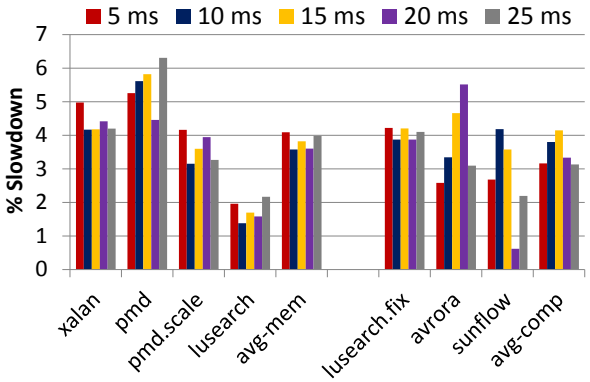


Figure 4.13: Per-benchmark slowdown for different quantum lengths. *The smaller quantum length of 5 ms incurs less overhead for the model while still achieving a slowdown close to the user-specified threshold.*

see increasing performance slowdowns as the quantum is increased, and sometimes their slowdown exceeds the user-specified 5%. lusearch.fix is not sensitive to quantum length, implying little or no phase behavior. We conclude that using a quantum length of 5 ms and a hold-off of 5 leads to performance that is, on average, close to the user's expectation. Using these parameters, the DEP+BURST model is active during only 20% of the benchmark's execution time.

To estimate the overhead of running DEP+BURST, we first note the number of epochs during the time we run DEP+BURST. We then use the latency of reading DVFS-related performance counters per epoch from prior work [35]. Our analysis show that the overhead of running DEP+BURST is less than 1% of the execution time of our benchmarks on average.

4.7.2 Case Study 2: Minimizing Full System Energy

To demonstrate the robustness of our energy manager to different optimization targets, we perform another case study, this time optimizing total system energy, i.e., the sum of the energy consumed by both the processor and DRAM. The optimal system energy is not obtained by running the processor at the lowest frequency, since as the processor frequency is lowered, the energy consumed by DRAM becomes the dominant factor. To optimize total system energy, the energy manager estimates the energy consumption at all the available DVFS states at the end of each profiling quantum, using the predictions of DEP+BURST for estimating the execution time, T'. The optimal frequency is the one which results in the lowest energy consumption. To estimate the total energy at a target DVFS setting (v', f'), when running at a base DVFS setting (v, f), we perform the following steps:

1. We scale the static power of the processor (processor-p-static) by a factor v/v' to estimate the processor-p-static at (v', f') [22].
2. We collect the estimated execution time, T', at f' from DEP+BURST. We multiply T' by the estimated processor-p-static to get the estimated static energy of the processor (processor-e-static).
3. We estimate the dynamic energy of the processor (processor-e-dynamic) as the sum of the dynamic energy of individual cores. The dynamic energy of each core at (v', f') is estimated by multiplying the energy consumed by the core at (v, f) with the factor, $(v/v')^2$, similar to [44].
4. The static energy of DRAM (dram-e-static) at (v', f') is estimated as T' multiplied by the static power consumed by the DRAM at (v, f).
5. For the number of DRAM requests seen in the previous quantum, we obtain the dynamic energy consumed by DRAM (dram-e-dynamic) from McPAT. Since a change in execution time does not impact the number of DRAM requests (only the request rate), we use the value obtained from McPAT as an estimate of the dynamic energy consumed by DRAM at the target frequency.
6. Finally, the total estimated energy at (v', f') is the sum of the estimated processor-e-static, processor-e-dynamic, dram-e-static and dram-e-dynamic.

Figure 4.14 shows the per-benchmark reduction in energy consumption obtained from different executions: running at the lowest frequency (1 GHz); running each benchmark multiple times offline, each time statically setting the frequency, and choosing the optimal energy consumption (Static-Opt); and dynamically adjusting the frequency at the end of each quantum using the above steps (Dynamic). Our baseline is the energy consumption obtained by running the entire benchmark at 4 GHz. We simulate a DDR3 DRAM main memory based on specifications from Micron [91].

First, some benchmarks experience an increase in energy from running at 1 GHz. Running at 1 GHz increases the execution time, which in turn increases the DRAM static energy. On average, running at 1 GHz reduces the energy consumption by only

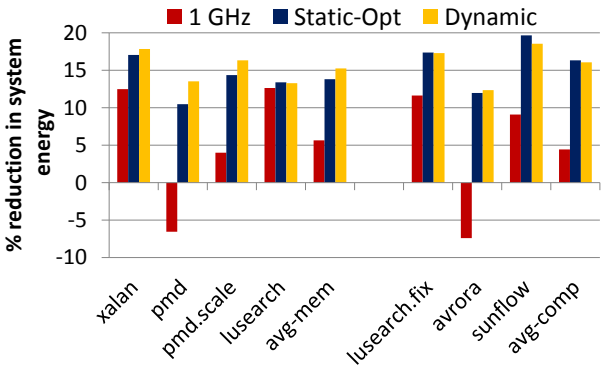


Figure 4.14: Per-benchmark reduction in energy consumption when the energy manager optimizes for total system energy. *Our energy manager achieves reduction in energy consumption that is comparable to or better than the optimal reduction in energy consumption obtained statically.*

5%. On the other hand, Static-Opt provides a higher reduction in energy consumption for all benchmarks. The average energy reduction is 15%, and the maximum reduction is 20% for sunflow. Since this case study does not impose a performance constraint, even the compute-intensive benchmarks benefit greatly from DVFS because reducing the processor voltage and frequency results in a quadratic drop in the dynamic energy consumed by the processor (at the expense of performance).

Next, we observe that our dynamic energy manager delivers a reduction in energy consumption on par with or better than Static-Opt. The average reduction in energy consumption is 15.6% with a maximum reduction of 18.5% for sunflow. For three benchmarks, including xalan, pmd, and pmd.scale, our proposed energy manager achieves a higher reduction in energy consumption than Static-Opt. Unlike Static-Opt, our dynamic manager is able to exploit phase behavior. Having an accurate performance predictor is necessary to optimize the full system energy consumption of multithreaded managed language applications.

In this section, we considered only the energy consumed by the processor and the DRAM. Other components such as the cooling unit, motherboard etc., also contribute to the system energy. Our energy manager can be easily extended to take into account any of these non-scaling components of system energy. It should be noted that if the increase in execution time due to lowering the processor's frequency leads to an increase in the total system energy - because the energy consumed by the other components offsets the reduction in the processor's dynamic energy consumption - our energy manager will run the processor at the highest frequency.

4.8 Summary and Interpretation

Accurate performance predictors are key to making effective use of dynamic voltage and frequency scaling (DVFS) to reduce energy consumption in modern processors. Multithreaded managed applications are ubiquitous yet prior work lacks accurate DVFS

performance predictors for these applications. In this work, we propose DEP+BURST, a novel performance prediction model to accurately predict the performance impact of DVFS for multithreaded managed applications. DEP decomposes execution time into epochs based on synchronization activity. This allows DEP to accurately capture inter-thread dependencies, and take the critical threads into account across epochs. BURST identifies critical store bursts and predicts their impact on overall performance as the frequency is scaled.

Our experimental results with multithreaded Java applications on a simulated quad-core processor report an average absolute error of 6% when predicting from 1 GHz to 4 GHz, and 8% when predicting from 4 GHz to 1 GHz using DEP+BURST, which is a substantial improvement over prior work. We demonstrate the usefulness of DEP+BURST by integrating it into an energy manager that 1) reduces the processor's energy by sacrificing a user-specified amount of performance, and 2) optimizes total system energy. For 1), with a user-specified slowdown of 5% and 10%, the energy manager is able to reduce energy consumption by 13% and 19% on average for a number of memory-intensive benchmarks. We show that our energy manager is robust to coarser frequency step settings, and incurs negligible execution time overhead. Finally, for 2), our energy manager demonstrates 15.6% total system energy consumption reduction on average.

This work opens up a new approach to build performance predictors. Hardware provides the necessary performance counters to keep track of microarchitectural events such as loads and stores. Software uses the semantic information in the runtime environment such as, e.g., inter-thread dependences, to predict the performance for different hardware parameters. Our focus in this work is predicting the performance impact of DVFS. More challenging is to predict the performance impact of changing core types on multithreaded managed applications. We leave it to future work.

This work is the first to propose a DVFS performance predictor for managed multithreaded applications. Several directions for future work are possible. Fine-grained dependencies between threads, such as those resulting from shared critical sections, could change at the target frequency. When this happens, DEP mispredicts the execution time at the target frequency. Efficiently dealing with mispredictions could improve the accuracy in meeting the user-specified slowdown thresholds, further reducing energy consumption. Another avenue for future work would be to explore per-core DVFS, as opposed to our current implementation that changes the frequency setting of all cores running a multithreaded application. DEP needs modifications to predict the performance impact of per-core DVFS. What is even more challenging is identifying the threads whose frequency change would result in the largest reduction in energy consumption. Finally, investigating the performance impact of DVFS when there is contention for either bandwidth or shared cache capacity, is a direction for future work.

Chapter 5

Kingsguard: Write-Rationing Garbage Collection for Hybrid Memories

5.1 Introduction

DRAM manufacturing complexity is shrinking supply and increasing main memory cost. Recent semiconductor analyses show that the DRAM price per gigabit increased by 50% between 2017 and 2018, whereas the year-over-year bit volume growth continued to decline [69, 58]. Main memory supply trends are especially worrisome as emerging applications have an insatiable desire for memory. Researchers have explored Non-Volatile Memory (NVM) technologies to expand main memory capacity [81, 82, 103]. Expecting DRAM supply shortages, Facebook is also building systems with NVM [42].

The most promising NVM technology, Phase Change Memory (PCM) [75, 81, 82, 103], offers five advantages: byte-addressability, high density, scalability (capacity), low standby power, and non-volatility, but four shortcomings: high access latency, write latencies exceed read latencies, high write energy, and low write endurance. Although improvements in PCM manufacturing technology are reducing latency [62, 95], it comes at the expense of write energy and endurance (lifetime). Endurance is the biggest challenge because each write changes the material form [21] and has thus far prevented NVM's uptake.

Prototype PCM hardware has an endurance of 1 million (M) to 100 M writes [81, 7]. This wide range results from: (1) the tradeoff between write speed and endurance, and (2) the properties of PCM materials. Prior architecture, operating system (OS), and programming language optimizations redirect and eliminate writes to improve lifetimes [20, 48, 81, 82, 103, 105, 125]. In particular, hybrid memories combine DRAM and PCM technology, seeking the best of both approaches [81, 82, 103]: (1) DRAM hides the high access latency of PCM by buffering frequently accessed pages,

and (2) hardware and the OS diffuse PCM writes with *wear-leveling* and reduce writes with page migration [20, 81, 82, 84, 103, 105, 125]. Wear-leveling moves pages and lines in pages to distribute writes uniformly. Page migration reactively places highly mutated pages in DRAM and read-mostly pages in PCM.

An open question for NVM memory systems is if modern applications can use NVM directly or if they require changes to runtimes and their programming models. Figure 5.1 explores this question with measurements of PCM lifetimes when executing Java applications. (Section 5.5 describes our experimental methodology.) These lifetimes motivate hybrid memories and including software support. The figure presents average lifetimes of a 32 GB PCM-only system for three different PCM endurance levels reported and used in prior work [81, 103, 101, 100, 84, 100, 21]. A 32-core PCM-only system with 32 GB of main memory and an endurance of 30 M writes per cell would wear out in 4 years, even with line write-back and wear-leveling [103, 101, 100]. Because lifetime is a linear function of writes, increasing endurance to 100 M per cell would improve PCM's lifetime to 13 years. However, running the Java application with the highest write rate would wear out a 32 GB PCM memory in less than 5 years. With current PCM endurance levels, a pure PCM memory system is thus impractical. For a 32 GB PCM-only system to last 15 years across a range of applications and endurance levels, write rates need to reduce by at least an order of magnitude.

In this chapter, we show that specializing the Java runtime system results in a promising and practical approach for using hybrid memories. Limiting changes to the runtime system, rather than requiring changes to programming models and applications, will ease adoption of hybrid memories.

We introduce the design and implementation of a new class of *write-rationing* garbage collectors that reorganize objects to limit writes to PCM in hybrid main memories, while still utilizing PCM capacity. We exploit the managed runtime implementation for languages such as Java, C#, JavaScript, and Python. Prior work either manages coarse-grained pages or allocation sites in C++ programs [125, 131] or profiles allocation sites ahead-of-time in a Java managed runtime, optimizing performance, but not the liftetime in hybrid memories [124]. In contrast, write-rationing collectors move and monitor individual objects in managed runtimes.

A detailed analysis of write behaviors in Java applications motivates our work. Figure 5.2 presents writes in an instrumented generational collector as a function of object age: young (nursery) versus mature space objects. 26% to 99% of writes occur to nursery objects, averaging 70%. The results are consistent with prior measurements [108, 132] and exhibit a wide range of behaviors. 2% of mature objects capture 81% mature object writes. 94% of all writes fall in one of these two categories. Our designs exploit the correlation between writes and object demographics (age) and detect the small fraction of mature objects that incurs most writes.

We introduce two write-rationing collectors that *guard* (i.e., Kingsguard) PCM from writes. They organize heap memory in DRAM and PCM virtual memory, directing the OS explicitly. The *Kingsguard-nursery* (KG-N) collector allocates new objects in a DRAM nursery, since they incur between 25% and 98% of writes, and then promotes all nursery survivors to a PCM mature space.

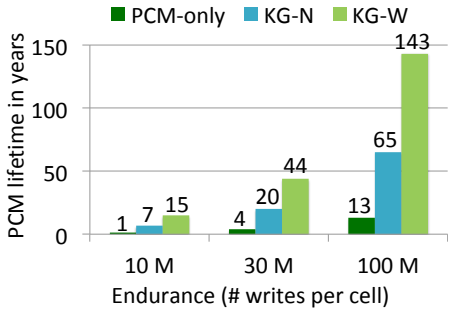


Figure 5.1: PCM-only is impractical. 32 GB lasts only 4 years on average with 30 M writes per cell and hardware line wear-leveling in simulation. The proposed KG-N and KG-W write-rationing garbage collectors manage DRAM and PCM, extending PCM's lifetime to practical levels.

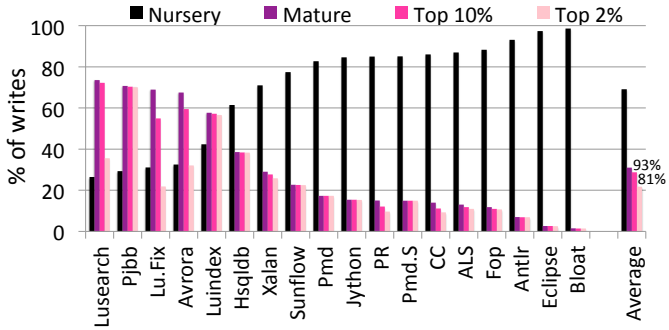


Figure 5.2: Nursery objects incur 70% of writes and mature objects incur 30% on average. The top 10% of written mature objects incur 93% of mature writes and the top 2% incur 81%.

Kingsguard-writers (KG-W) adds fine-grained monitoring and per-object placement of mature objects. It also uses a DRAM nursery, but promotes nursery survivors to a DRAM *observer* space. The Java Virtual Machine (JVM) tracks all mature object writes with a write-barrier [38, 46, 111, 128]. Observer space collections copy objects with zero writes from the observer space to the PCM mature space and copy any written objects to the DRAM mature space, using past writes to predict future writes. Kingsguard-writers promotes most observer space survivors (90%) to PCM memory, thus exploiting its capacity. When it detects written objects in PCM, it moves them back to DRAM. KG-W also includes optimizations for large objects and object meta-data.

Because hybrid memory systems are not available, we use cycle-level multicore simulation and hardware measurements for evaluation. We find that KG-N and KG-W improve PCM lifetime by 5× and 11×, respectively on our simulated Java applications. KG-W needs 16 MB of DRAM to achieve these lifetimes. We compare to state-of-the-art OS write partitioning (WP) [131]. WP consumes about the same amount of DRAM, but has 3× more writes to PCM than KG-W. Even though memory accesses to PCM are slower, Kingsguard-nursery reduces the energy-delay product by 36%

over a DRAM-only system and 33% over a PCM-only system. Kingsguard-writers adds 5% average overhead to monitor nursery survivors in the observer space and to copy objects between spaces. Kingsguard-writers reduces the energy-delay product by 32% on average over DRAM-only and by 29% over a PCM-only system. Compared to Kingsguard-nursery, Kingsguard-writers thus trades some overhead and DRAM to significantly improve PCM lifetime.

In summary, this chapter makes the following contributions:

- an empirical characterization of Java applications that motivates hybrid memories and fine-grained object placement;
- the design and implementation of write-rationing garbage collectors that manage hybrid memories, minimizing PCM writes while maximizing the use of their capacity;
- Kingsguard collectors that explicitly allocate and move objects into DRAM and PCM heap spaces based on their demographics and write behavior;
- execution and simulation results that show these collectors exploit PCM capacity while substantially improving PCM lifetimes and energy by reducing writes as compared to prior OS and hardware approaches; and
- a practical approach to exploit hybrid memories that requires no new OS or hardware support.

5.2 Related Work

This section discusses work related to PCM hardware, and OS and runtime approaches to manage hybrid DRAM-PCM memory systems.

Hardware and OS support for PCM. The two predominant approaches for improving PCM lifetime are making writes more uniform over the PCM capacity, called *wear-leveling*, and reducing the number of writes. Wear-leveling of pages allocates and moves pages to distribute writes uniformly through memory. Wear-leveling of lines within a page remaps lines to distribute writes uniformly on each page. Prior work proposes a number of wear-leveling approaches [101, 102, 110]. We use line wear-leveling from Qureshi et al. [103] as our baseline hardware.

Prior work proposes hardware and OS techniques for hybrid DRAM-PCM memories that monitor and move pages to reduce PCM writes [20, 48, 81, 82, 84, 103, 105, 125]. Their two main drawbacks are (1) they are reactive, and (2) they work at the page granularity. None of these systems considers reorganizing objects on pages to create pages of read-mostly objects and pages of mutated objects, as we do.

We implement OS Write Partition (WP) by Zhang et al. [84, 105, 131] and find that our write-rationing garbage collectors decrease writes by 3 $\times$ more than WP (see Section 5.6.1). WP treats DRAM as a partition for highly mutated pages, which it

identifies using a ranking scheme. The ranking scheme is a variation of the widely used Multi Queue algorithm for managing OS buffer caches [133]. The OS places a new page in PCM first. The memory controller then counts writes to each physical page and tracks pages in queues ordered by power of 2 writes. When a page incurs a threshold number of writes, the memory controller promotes that page to a higher ranked queue: at 2^n writes, the OS promotes the page to the queue with rank n. The OS periodically migrates pages in the highest-ranked queues to DRAM. Subsequent work builds upon WP, adding performance optimizations, but does not change lifetime [84, 105]. Since write-rationing collectors optimize lifetime, we compare to WP and find our approach incurs substantially fewer PCM writes.

Memory management and garbage collection. The closest related work also uses the managed runtime, but optimizes for performance in hybrid memories, as opposed to lifetime [124]. It performs an offline profiling phase to identify object allocation sites for cold (rarely read or written) and hot old objects. It places all nursery objects in DRAM. It promotes nursery survivors according to their tag, moving hot objects to DRAM and cold ones to PCM. Our work optimizes for a different goal — PCM lifetime. Our work requires no ahead of time profiling, which suffers when inputs do not match the profile. It dynamically monitors individual object writes in the observer space to manage writes to PCM.

A similar offline-profiling approach for C programs finds allocation sites that produce highly mutated memory and ones that produce read-mostly memory, modifying allocations sites to specify PCM or DRAM [125]. C semantics limit this approach because objects cannot move. Our work exploits managed language semantics to monitor and move objects, making fine-grained per-object decisions and correcting them if need be, regardless of allocation site.

To tolerate PCM line failures in a page while running managed language applications, Gao et al. introduce new hardware and OS approaches to mask defective lines, but when lines fail, they expose defective lines in page maps to the garbage collector [48]. The OS provides this map to the runtime and copies of data during a runtime failure, which prevents using or losing failed lines by the collector. They did not consider hybrid memories.

Prior work observes that nursery collections leave behind written cache lines full of dead objects and propose cache scrubbing instructions that mark these lines as dead after a nursery collection, preventing writes to DRAM [108]. Our approach is complementary. It exploits this observation to protect PCM from writes of highly mutated nursery objects. Complementary approaches also include dividing the heap into hot and cold regions to better manage DRAM energy consumption [63], and data structure-aware heap partitioning to improve lifetimes, locality, and region-based memory management [94].

5.3 Background

Chapter 2 discusses our assumed memory and storage hierarchy and also provides background on the Immix mark-region generational garbage collector (GenImmix) [18] on which we build. Here, we discuss additional background relevant to Kingsguard collectors.

Immix: a generational mark-region collector. We modify GenImmix, the default best-performing collector in Jikes [18], to create Kingsguard collectors. GenImmix uses a copying nursery and a *mark-region* mature space. The mark-region mature space consists of a hierarchy of two regions: blocks and lines. Blocks are multiples of page sizes and consist of multiple lines. Lines are multiples of cache line sizes. Objects may cross lines, but not blocks. Bump pointer object allocation is contiguous in the nursery. (Contiguous allocation is known to outperform free-list allocators due to its locality benefits [12, 18, 61].) Filling the nursery triggers a collection, which copies nursery survivors contiguously into free lines within blocks in the mature space. Filling the mature space triggers a full heap collection. Immix reclaims the mature space at a line and block granularity by marking lines and blocks live as it traces and marks live objects. Subsequent mature allocation bump-point allocates first into contiguous free lines in partially free blocks and then into completely free blocks. Allocation and reclamation use per-thread allocators and work queues to deliver concurrency and scalability. The per-thread allocators obtain blocks (partially and completely free) from a global allocator.

We use the default settings for Immix, including the maximum object size (8 KB), line size (256 bytes), and block size (32 KB). These settings match the Immix line size to the PCM line size. Immix tailors the heap representation to match the hardware memory system for performance, but it also matches the needs of PCM memory management for detecting and tolerating line failures, as Gao et al. [48] show.

5.3.1 Write Barriers

For correctness, generational collectors use write-barriers to record pointer references from mature generation objects to nursery objects in order to collect the nursery independently [121, 128]. Valid remembered references serve as roots during a minor collection. The collector updates source objects in the mature space with the new locations of referent objects, now relocated in the mature space. The compiler inserts a few lines of code on every write that records (*remembers*) references when an application write installs a pointer from a mature object to a nursery object. Generational collectors organize all nursery objects on one side of a *boundary* and all mature objects on another, so the write barrier simply tests if pointers cross the boundary from mature to young. Prior work shows these references are relatively rare, but even when more frequent, the cost of barriers is low, ranging from less than 1% to 3% on modern hardware [128].

5.3.2 Large Objects

Jikes RVM manages objects larger than the 8 KB threshold separately, allocating them directly into a non-copying large object space, and uses a *treadmill* to avoid copying them [56, 67]. A treadmill consists of two doubly-linked lists that store all references to large objects. During collection, live traced references are removed from one doubly-linked list and snapped to another. The collector then reclaims the large objects reachable from any unsnapped references. The cost of using a treadmill is high due to storing all the references to each large object, which is only justified because it eliminates marking large objects.

5.3.3 Object Metadata

In addition to application writes, the JVM and collector also generate writes. In particular, they write object metadata during allocation and collection. Object metadata includes an object's type, layout, and liveness information. The liveness information is often stored in a header word next to the object. Garbage collectors write to metadata when they mark objects live and they write to objects directly when they update their references after copying objects. When GenImmix marks a mature object live, it also writes block and line bits, stored separately from the object. Because marking live mature objects in PCM generates a lot of PCM writes when liveness is stored in the object header, the KG-W write-rationing collector includes an optimization that eliminates these writes by storing object liveness metadata in DRAM space separate from the objects.

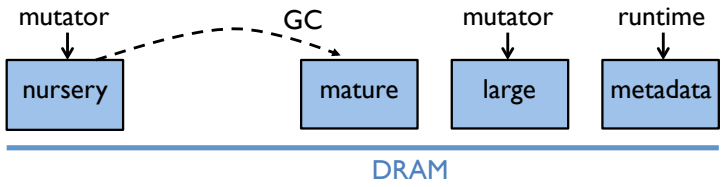
5.4 Write-Rationing Garbage Collection

This section presents the design of write-rationing garbage collectors that seek (1) all the performance advantages of high-performance garbage collection, (2) to maximize the use of PCM for its scalability properties, and (3) to limit writes to PCM to extend its lifetime. These collectors limit write traffic to PCM significantly compared to PCM-only memory by judiciously placing highly mutated objects in DRAM.

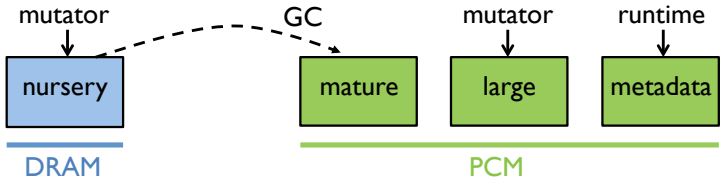
Our baseline memory systems contain DRAM-only or PCM-only memory and use the generational Immix (GenImmix) collector (see Chapter 5.3). Figure 5.3(a) illustrates this baseline generational heap organization with DRAM-only (and PCM-only) memory. The mutator allocates objects into the nursery and large object space. The collector copies surviving nursery objects to the mark-region mature space. The JVM uses a metadata space. Mature and large object space collection is non-moving (not shown).

5.4.1 Kingsguard-nursery for Hybrid Memory

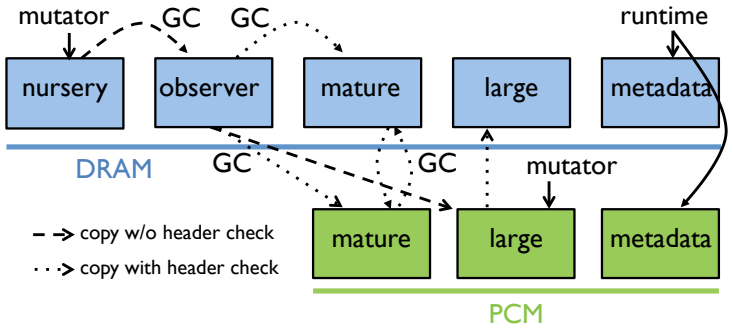
As motivated by the data in Figure 5.2, a promising strategy for limiting writes to PCM in hybrid memories is placing nursery objects in DRAM and all other objects in



(a) Homogeneous DRAM-only system



(b) Kingsguard-nursery hybrid memory



(c) Kingsguard-writers hybrid memory

Figure 5.3: Main memory heap organizations (not to scale).

PCM. We call this collector Kingsguard-nursery (KG-N). It also puts nursery survivors (mature), large objects, and JVM metadata in PCM memory. Figure 5.3(b) illustrates how Kingsguard-nursery maps the baseline heap organization onto hybrid memory. The system requests DRAM memory from the OS for the nursery, where objects are freshly allocated, and requests PCM memory from the OS for everything else. Requests to the OS are at the page granularity (4 KB). This heap organization is appealing because it maximizes the number of objects that reside in PCM and it requires minimal changes to the VM and garbage collector.

Compared to a PCM-only system, Kingsguard-nursery eliminates the nursery writes to PCM (70% of all writes), saves energy, and increases PCM's lifetime. Because applications also write to mature objects, many applications will still wear out PCM with this approach.

5.4.2 Kingsguard-writers for Hybrid Memory

Kingsguard-writers (KG-W) adds new heap regions and mechanisms to further limit writes to PCM by monitoring and placing individual objects in DRAM or PCM. Kingsguard-writers also allocates all new objects in a DRAM nursery. In addition, it creates a new DRAM *observer* region for nursery survivor objects where it monitors their writes. It then chooses on an individual object basis to put mutated objects in a mature DRAM space and unwritten objects in a mature PCM memory space. Of course these objects can have subsequent writes, so we call them *read-mostly* objects. KG-W has two large object spaces: one in DRAM and one in PCM. KG-W initially places large objects in the nursery if space is available, since a surprising number die quickly, or otherwise allocates them directly to PCM memory. KG-W monitors small and large objects in PCM, and when it detects written objects, it moves them to their corresponding space in DRAM during the next collection. Figure 5.3(c) illustrates Kingsguard-writers' heap organization.

The Observer Space

Rather than monitoring all objects for writes, we restrict object monitoring to mature objects that survive at least one nursery collection. Because nursery objects are rapidly mutated, monitoring them would incur high overhead. Even when many survive, zeroing, initialization, and data structure creation produce a flurry of writes. Table 5.4 reports nursery survival rates of 17 % on average, as low as 0.001 %, and as high as 66%.

KG-W adds a new DRAM generation for all nursery survivors, called the *observer* space. While objects reside in the observer space, KG-W monitors all writes and marks a bit when objects are written. The observer space is a contiguous region and uses bump-pointer allocation. We make the observer space twice as large as the nursery and trigger an observer space collection when it is full. An observer collection thus results in pause times longer than nursery collections, but shorter than full heap collections. An observer collection moves live unwritten objects to the PCM mature space and live written objects to the DRAM mature space.

The observer space achieves two goals. (1) It gives objects more time to die, so fewer objects are even candidates for PCM memory. (2) If the system detects a write to an object while it resides in the observer space, the collector never moves it to PCM, because it uses writes to objects in the observer space as a predictor of future writes.

KG-W copies all surviving nursery objects to the observer space instead of the mature space and collects the observer space more frequently than the mature space in the baseline system. KG-W reserves a small amount of room in the observer space into which it copies surviving nursery objects during an observer space collection. It sizes this room using recent nursery survival rates. Some surviving objects will therefore die soon afterward in the observer space. Kingsguard-nursery would copy these objects to the PCM mature space, which creates dirty dead cache lines that are likely to be written back once the mutator resumes execution. Kingsguard-writers avoids these useless writes to PCM by using the observer space to avoid tenuring garbage.

In Hotspot [38, 98], a survivor space in the young generation serves a similar purpose of giving objects more time to die, but objects reside in this space only until the next nursery collection. Other collection strategies also seek to avoid tenured garbage [121, 115, 126]. All these approaches are orthogonal to our work.

Monitoring and Write Barriers

An *observer collection* includes the nursery and observer space, in isolation of other spaces, to make timely promotions from the observer space to PCM. To ease the recording of references into the nursery and observer space required to collect them independently of the mature space, we colocate the nursery and observer space on the same side of a virtual address space boundary. We thus can and do use the same fast boundary write-barrier as used by a standard generational collector. Figure 5.4 shows our modified reference write-barrier in Jikes RVM.

In generational collections, pointers from outside the nursery into the nursery are added to the root set for nursery collections. Similarly in KG-W, pointers from outside the nursery and observer spaces into those spaces are added to the root set for observer collections. Lines 7 to 12 in Figure 5.4 shows the code KG-W executes for remembering the pointers from outside the nursery and observer spaces. This part of the write-barrier is the same as a standard generational write-barrier. The later part of the barrier monitors reference and primitive writes to objects.

```
1  @Inline
2  public final void objectReferenceWrite(
3    ObjectReference src,
4    Address slot,
5    ObjectReference tgt)
6  {
7    if(!inNursery(slot) && inNursery(tgt)) {
8      remset.insert(slot);
9    }
10   if(!inNurseryOrObservers(slot) && inNurseryOrObservers(tgt)) {
11     remset_observers.insert(slot);
12   }
13   if(!inNursery(src)) {
14     Object o = ObjectReference.fromObject(src);
15     Address a = o.toAddress();
16     a.store(Word.one(), EXTRA_WORD_OFFSET);
17   }
18   Magic.setObjectAtOffset(src, slot, target);
19 }
```

Figure 5.4: Our modified reference write-barrier. Lines 10 to 17 show the extra code KG-W executes on each reference write.

The main benefit of the observer space is to monitor writes to objects and use their behavior to determine on a fine-grained object level whether to put an object in the DRAM or in the PCM mature space. Our analysis shows that 81% of writes to non-nursery objects happen to 2% of objects, as shown in Figure 5.2. Objects written a number of times are, therefore, likely to be written again.

To monitor observer space writes, we use the write-barriers on (1) references and (2) primitives that are provided by MMTk [12]. All of our systems must monitor references (pointers) to collect the nursery and observer spaces independently, so additionally monitoring writes to references in the observer space incurs little additional overhead. On the other hand, monitoring primitives, writes to all other values, has higher overhead because primitive writes are more common than reference writes and are not necessary for collecting independent regions correctly. Reference writes, often, but not always, predict primitive writes. We show the performance-accuracy tradeoff of using the two types of barriers in Section 5.6.2.

We modify the generational write-barrier to monitor writes to references and primitive fields outside the nursery space. To remember written objects, we add a *write word* in each object header. When the program writes an object in the observer space, the write-barrier sets a bit in this header word as shown in lines 13 to 17 in Figure 5.4. We add a header word because GenImmix uses all the existing object header bits [18]. A careful re-design or disabling some Immix features, such as pinning, could steal a bit instead. Since we have an entire word, the barrier could record the number of writes. We leave reducing this extra header space and counting writes for future work. We use one of the remaining extra header bits for the metadata optimization presented below.

Mature DRAM and Mature PCM Spaces

During an observer collection, Kingsguard-writers checks the write bit and then copies all written objects from the observer space into the *mature DRAM* space, and the remaining objects to the *mature PCM* space. Figure 5.3(c) shows this selective copying with dotted lines.

KG-W monitors all writes in the mature DRAM and mature PCM spaces. During whole-heap collections, KG-W moves objects in the mature DRAM space whose write bit is zero, and we therefore predict will not be written, to the mature PCM space. This step adds copying work to more fully exploit the capacity of PCM. Similarly, when KG-W detects a written object in mature PCM, it copies the object to the mature DRAM space, and resets its write bit to zero. This step adds copying work to limit future writes (predicted by the past writes) to PCM by this object.

We trigger a full heap collection when the combined mature DRAM and mature PCM spaces run out of space, based on the heap size used. Kingsguard-writers does not limit the amount of DRAM space. However, because of the high mortality rates in both the nursery and observer spaces, very few objects are put into the mature DRAM space, which is between 26 MB and 40 MB for our applications. By design, the PCM portion of the mature generation contains objects that are likely long-lived and infrequently written. For applications that mostly create small objects, mature PCM is the largest portion of the heap. We discuss the amount of DRAM and PCM used per benchmark in Section 5.6.

Large Object Optimization (LOO)

Most collectors manage large objects separately because they are costly to copy. In hybrid memories, if they are frequently written, they are also costly in writes, degrading PCM lifetime. We achieve the best of both worlds by creating a large object space in DRAM and in PCM. To exploit the capacity of PCM, KG-W initially puts large objects directly in the large PCM space. If the mutator writes to large objects when they are in the large PCM space, we move them to the large DRAM space during the next major collection. We modify the non-moving treadmill data structure used for large objects to handle moving objects. When copying from large PCM to large DRAM, objects are unsnapped from the former treadmill's linked list and snapped on to the latter space's treadmill. Because copying large objects incurs a high overhead, once a large object is copied to DRAM, we never move it back to PCM.

We find empirically that large objects often follow the weak-generational hypothesis, i.e., they die quickly. Therefore we perform a dynamic optimization (LOO) to place some large objects in the nursery first to give them a chance to die and to avoid allocating large objects that die quickly in PCM. If a large object survives a nursery collection, we copy it to the observer space. If it survives an observer collection, KG-W copies it directly to the large PCM space, without consulting the write bit, to leverage the capacity of PCM.

Allocating large objects in the nursery should be done with caution, to ensure space for small objects. KG-W dynamically monitors the allocation rate to choose whether to devote part of the nursery to large objects or not. If at the end of a nursery collection, the allocation rate in the large PCM space is faster than the nursery's allocation rate, then we enable this optimization. The allocator allocates large objects less than half of the remaining nursery size in the nursery, and otherwise allocates them in the large PCM space. This technique gives large objects time to die and for arrays of references, gives any referent objects time to die as well. Section 5.6 shows this optimization saves a lot of allocation and writes to PCM, thus improving its lifetime. This optimization trades some copying overhead to limit writes to the large PCM space.

Metadata Optimization (MDO)

As mentioned in Section 5.3, garbage collectors require metadata to track object liveness. GenImmix stores the mark state of objects in their headers, which would result in writes to all live mature objects in PCM memory when collecting the whole heap. Updating a single byte in the header of each live object in PCM would result in writing back one cache line for every live object in PCM on every major collection. KG-W thus performs an optimization (MDO) to decouple the mark state metadata from the PCM object.

KG-W stores the mark states of objects in mature PCM in a separate metadata region in DRAM (shown in Figure 5.3(c)). Using the Immix allocator, the PCM mature space reserves new space 4 MB at a time. When this happens, KG-W allocates a table in DRAM for the mark state of objects in that 4 MB region. The table size depends on the number of objects that fit in 4 MB. Object sizes vary from 4 bytes (a header with no payload) to 8 KB (the biggest small object). Accounting for the

Table 5.1: Collector configurations. Large Object nursery allocation (LOO) and PCM metadata in DRAM (MDO) are dropped in two configurations.

Configurations	monitor writes	metadata in DRAM	LOO in nursery
KG-N: Kingsguard-nursery	✗	✗	✗
KG-W: Kingsguard-writers	✔	✔	✔
KG-W–LOO	✔	✔	✗
KG-W–LOO–MDO	✔	✗	✗

smallest object would incur a 25% DRAM overhead. We empirically find that most objects are larger than 16 bytes. We therefore reserve 262 KB for the mark state table for each 4 MB region of PCM, incurring a 6.25% overhead for storing the mark states separately. We use this table for all objects over 16 bytes. For objects 16 bytes and smaller, we mark them small in the write word in the object's header and use the normal mark bit in the header instead of the mark state table. For fast access, we store the address of the mark state table at the beginning of each 4 MB PCM region. To calculate the mark state address of a PCM object, we add the object offset in the 4 MB region to the starting address of the table. When the collector frees a 4 MB region in PCM, it also frees the DRAM space reserved for the mark state table. Section 5.6 shows that this optimization reduces the collector's writes to PCM for highly allocating applications.

5.5 Experimental Methodology

This section describes our experimental methodology, including our JVM, applications, collector configurations, hardware, architectural simulator, and memory models.

5.5.1 Software

Garbage collectors and configurations. We compare to Jikes RVM's default stop-the-world generational Immix collector [18], with DRAM-only and PCM-only heaps. We use the default settings for maximum object size (8 KB), line size (256 bytes), and block size (32 KB). We explore four write-rationing garbage collectors shown in Table 5.1: Kingsguard-nursery, Kingsguard-writers, and two variants that exclude the Large Object Optimization (LOO) and the Metadata Optimization (MDO) to tease apart their impact.

The nursery size impacts performance, pause time, and space efficiency [6, 12, 121, 132]. Our default configurations use a 4 MB nursery and a fixed-size maximum heap size of 2× minimum live size for each application, following prior work [1, 18, 108, 132, 94]. Fixing the heap size fairly controls the space-time tradeoffs that different collectors make. We set the default observer space size at 8 MB. We empirically find

Table 5.2: Simulated system parameters.

Component	**Parameters**
Processor	1 socket, 4 cores
Core	4-way issue, 4.0 GHz, 128-entry ROB
Branch predictor	hybrid local/global predictor
Max. outstanding	48 loads, 32 stores, 10 L1-D misses
L1-I	32 KB, 4 way, 4 cycle access time
L1-D	32 KB, 8 way, 4 cycle access time
L2 cache	256 KB per core, 8 way, 8 cycle
L3 cache	shared 4 MB, 16 way, 30 cycle
Coherence protocol	MESI
Memory controller	FR-FCFS scheduling, line-interleaved
	mapping, closed-page policy
Memory bandwidth	12 GB/s
Memory systems	32 GB DRAM-only
	32 GB PCM-only
	Hybrid 1 GB DRAM + 32 GB PCM
Organization	8 1 Gb chips per rank
	1-8 ranks per DIMM, 1-4 DIMMs
DRAM parameters	45 ns read/write
	0.678 Watts read, 0.825 Watts write
PCM parameters	180 ns read, 450 ns write
	0.617 Watts read, 3.0 Watts write
	30.0 million writes per cell
	Fine-grained wear-leveling [103]
DRAM device	Micron DDR3 [91]

that sizing the observer space to be twice that of the nursery is the best compromise between tenured garbage and pause time. We explore other configurations, including larger nursery sizes (Section 5.6.2). Large nurseries reduce writes to mature objects, but are not sufficient to manage hybrid memories.

Java applications. We use 16 Java applications: 12 DaCapo [14], pseudojbb2005 (pjbb) [16], and 3 graphchi [78] applications. The graphchi applications are disk-based graph processing applications, including: (1) page rank (PR), (2) connected components (CC), and (3) ALS matrix factorization (ALS). For PR and CC, we use the LiveJournal online social network [85] as the input dataset. For ALS, we use the training set of the Netflix Challenge. We process 1 M edges using PR and CC, and 1 M ratings using ALS. We use the default datasets for DaCapo and pjbb2005. In addition to the original versions of lusearch and pmd in DaCapo, we use an updated version of lusearch, called lu.Fix (described in [129]), that eliminates useless allocation, and

an updated version of pmd, called pmd.S (described in [40]) that eliminates a scaling bottleneck due to a large input file.

5.5.2 Hardware and Simulation

Our evaluation uses both simulation of hybrid memories and execution on real hardware with DRAM memory because we lack access to systems with hybrid memories. We use simulation to evaluate Kingsguard collectors because simulation models PCM properties accurately. Because this is the first work to use garbage collection to mitigate PCM wear-out, we want to compare to a prior OS solution. We find it feasible to implement the OS solution in the simulator. We also use the simulator to obtain the energy efficiency results for memory systems with DRAM and PCM. Finally, the simulator helps us to gain insight into the sources of cache writebacks to main memory and thus the origin of PCM writes. Encouraged by the comparison to the OS approach and energy efficiency results, the next chapter uses an emulation platform to compare different write-rationing garbage collectors.

Due to limitations, the simulator unfortunately can execute only a subset of the benchmarks. Fortunately, these benchmarks cover the extremes and thus a wide range of write of behaviors. All our applications execute on real hardware. We furthermore configure the real hardware in various ways to match and validate many of the simulation results.

Hardware Platform

We use the Intel Nehalem-based IBM x3650 M2 with two Intel Xeon X5570 processors for hardware execution time measurements. Each Xeon processor has 4 cores. Although there are two sockets, we use one to limit non-determinism and to match the multicore simulator, for which it is only practical to run with at most 4 cores. Each core has a private L1 with 32 KB for data and 32 KB for instructions. The unified 256 KB L2 cache is private, and the 8 MB L3 cache is shared across all four cores on each socket. The machine has a main memory capacity of 14 GB.

Simulator

Because PCM is not commercially available, we modify a simulator to model hybrid memories. We use Sniper [26] v6.0, an x86 simulator because it is cycle-level, parallel, high-speed and models multicore systems. We use its most detailed cycle-level and hardware-validated core model. Prior work extended Sniper for managed language runtimes, including dynamic compilation, and emulation of frequently used system calls [108]. Because Sniper is a user-level simulator, we are only able to execute ten of the Java applications. We eliminate fop, luindex, and avrora from simulation results because of their low allocation rates (see Table 5.4). These limitations motivate our additional results gathered on actual hardware.

Memory system and processor architectures. We compare three main memory systems in the simulator: (1) a 32 GB DRAM-only system, (2) a 32 GB PCM-only system, and (3) a hybrid system with 1 GB DRAM plus 32 GB PCM. We model PCM with a base read latency of 4× the DRAM latency, and a write latency of 12× the DRAM latency [62, 81, 95]. Table 5.2 presents other key architecture, DRAM, PCM, and cache memory parameters, which we hold constant. We model a quad-core processor configuration similar to the Intel Haswell processor i7-4770K. Each core is a superscalar out-of-order core with private L1 and L2, and shared L3 caches.

Power and energy estimation. We use McPAT v1.0 [86] to model processor power consumption. We model DRAM power according to Micron's DDR3 device specifications [91]. PCM uses a 1 KB row buffer similar to DRAM. The remaining peripheral circuitry is also similar with one important distinction. When writing data from a row buffer to a PCM array, only the modified line is written back. PCM read operations do not require pre-charging due to their non-destructive nature and consume less energy than DRAM. The static power of PCM prototypes are negligible compared to DRAM [82]. Using latency and energy estimations of PCM prototypes from Lee et al. [81], we compute the average power to write a cache line to a PCM array as 3 Watts. When estimating PCM latency and power consumption, we assume the same technology node for DRAM and PCM, and the scaling model from Lee et al. [81].

PCM lifetime modeling. We estimate PCM lifetime using an optimistic analytical model from the literature [103, 62, 95]. Prior work demonstrates wear-leveling mechanisms for future non-volatile memories [20, 48, 81, 82, 103, 105, 125]. Therefore, we assume writes can be made uniform over the entire capacity of PCM. PCM memory lifetime in terms of years before failing is estimated as follows:

$$Y = \frac{S \times E}{B \times 2^{25}} \tag{5.1}$$

The size (S) of PCM main memory is 32 GB. We consider the PCM endurance (E) level used in prior work [100, 101]: 30 M writes per PCM cell. Finally, B is the write rate of an application during execution. Next, we describe our methodology for estimating the write rates of our applications on a 32-core machine.

Write rate estimation. Due to limitations in simulator scalability, we are unfortunately only able to simulate a 4-core system. To extend our simulation results to write rates for a 32-core machine, we first obtain write rates for the 4-core system in Table 5.3. We then measure on a real hardware platform how write rates scale as we increase the number of cores from 4 to 32. We multiply the observed scaling behavior by our simulated write rates to estimate the write rates on a 32-core system. Our 32-core system has two Intel E5-2650L processors. Each processor has 8 cores and each core is 2-way SMT. Each processor has a 20 MB last level cache. The machine has 132 GB of main memory.

Table 5.3: Measured scaling of and estimated write rates.

Benchmark	Normalized scaling factor (measured)	Write rate in GB/s (estimated)
Xalan	7.3×	8.5
Pmd	7.7×	3.1
Pmd.Scale	10.0×	7.0
Lusearch	5.0×	9.3
Lu.Fix	5.2×	7.0
Antlr	52.0×	19.0
Bloat	63.0×	24.0

To measure write rates on real hardware, we use the Processor Counter Monitor from Intel. To fully utilize the 32 cores on our system, we run 32 instances of the same single-threaded benchmark, and 8 instances of the multithreaded benchmarks. Table 5.3 shows the scaling factor and write rates with multiple instances of each benchmark normalized to running a single instance of the benchmark. A few benchmarks scale linearly with the increase in core counts, but for others, such as antlr and bloat, the write rates increase by more than an order of magnitude. These applications experience increased contention in the last level cache. Table 5.3 shows estimated write rates vary from 3.1 GB/s to 24 GB/s.

5.6 Results

Section 5.6.1 presents our simulation results which evaluate PCM lifetimes, write behavior, energy, and overhead. We compare DRAM-only, PCM-only, and hybrid systems with Kingsguard collectors. Our cycle-level simulator faithfully models the cache hierarchy found in real systems and wear-leveling hardware. Modeling caches is important because they absorb writes, and are thus the first line of defense in protecting PCM from writes. Modeling wear-leveling is important because by spreading writes to lines and pages evenly, it makes write rate the only necessary target for optimization.

Section 5.6.2 presents performance results on real hardware of all Kingsguard configurations. It includes statistics on how Kingsguard collectors organize the heap to influence PCM write traffic for 16 Java applications and write traffic to PCM measured in an architecture-independent manner.

Both sets of results show significant improvements in PCM lifetimes when using hybrid memories and our write-rationing garbage collectors. The hardware results confirm the simulation results and explore overheads and optimizations in more detail.

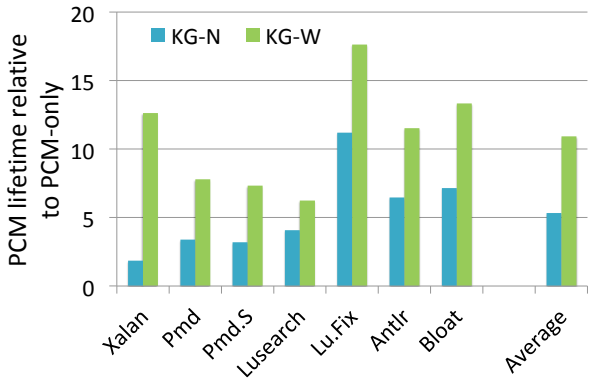


Figure 5.5: Kingsguard-nursery (KG-N) and Kingsguard-writers (KG-W) increase PCM lifetime.

5.6.1 Simulation Results

Lifetime

Figure 5.5 presents PCM lifetime improvements normalized to PCM-only using lifetime estimates from the models detailed in Section 5.5. On a 32 GB PCM-only system with an endurance of 30 M writes, line-level write back and wear-leveling, application lifetimes average 4 years, but are sometimes as low as 15 months for lusearch. Kingsguard collectors executing on a hybrid memory system deliver substantial lifetime improvements over PCM-only systems. KG-N improves lifetime on average by 5×. Individual benchmarks improve by 1.9× for xalan and up to 11× for lu.Fix. KG-W improves lifetime even more: 11× longer than a PCM-only system on average. Individual benchmarks improve by 6× for lusearch and up to 17× for lu.Fix. KG-W achieves these long lifetimes by minimizing writes to PCM memory, while using an average of only 16 MB of DRAM and at most 24 MB of DRAM for these applications (see Table 5.4 and Section 6.7.7).

Write Analysis

Figure 5.6 plots writes to PCM using the Kingsguard configurations from Table 5.1 normalized to PCM-only. While KG-N reduces writes to PCM by 81% on average, KG-W reduces writes by 91% compared to a PCM-only system. Leaving out the large object optimization (KG-W–LOO) and metadata optimization (KG-W–LOO–MDO) has a small overall impact, except for xalan. In xalan, the large object optimization reduces writes to large objects and, more surprisingly, writes to small objects to which the large objects point. While the effect on total writes is small with MDO, it does eliminate a lot of writes to PCM during major collections: 50% and 12% respectively for xalan and lusearch.

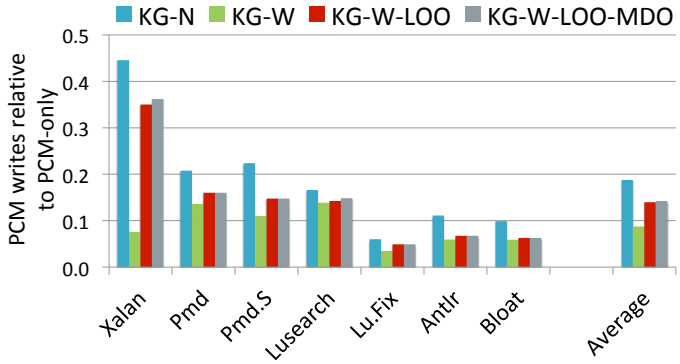


Figure 5.6: All Kingsguard configurations substantially reduce writes compared to the PCM-only baseline (1.0).

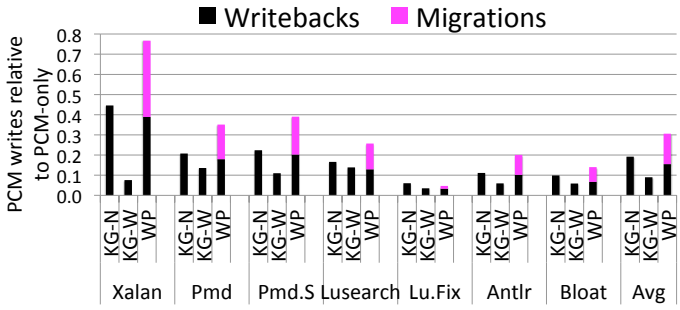


Figure 5.7: OS-managed write partitioning (WP) results in more writes to PCM than KG-N and KG-W.

Comparison to Write Partitioning (WP)

We now compare Write Partitioning (WP), the state-of-the-art OS technique for reducing writes to PCM [131, 133]. Our implementation, as described in Section 5.2, uses the recommended eight queues, an OS mapping time quantum of 10 ms, migrates pages in the four highest ranked queues to DRAM, and demotes all pages in DRAM to a lower ranked queue every 50 ms to optimize for phase behavior. We explore other configurations, but these parameters perform best for our workloads. Figure 5.7 plots PCM write reductions by KG-N, KG-W, and WP, normalized to PCM-only. *Writebacks* include writes by the application and collector. *Migrations* show writes due to WP migrating pages from DRAM to PCM. WP's reactive policy does eventually detect nursery pages as highly written, but this detection takes time. WP is effective at reducing application writes to PCM, but its migration policy moves pages from PCM to DRAM and back to PCM. For example, WP observes lots of writes when the default collector copies nursery objects to the mature space, which triggers WP to migrate pages from PCM to DRAM. Many of these pages incur few subsequent writes, so WP migrates them back to PCM. WP reduces writes to PCM by 69%, whereas KG-N and KG-W reduce writes by 81% and 91%, respectively. By using object demographics and fine-grained per-object monitoring, KG-W has over 3× fewer writes than WP.

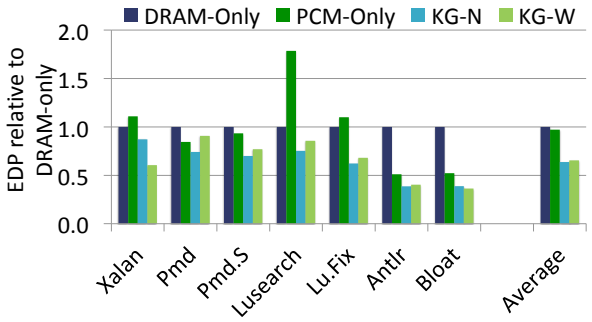


Figure 5.8: Kingsguard reduces the energy-delay product (EDP) compared to DRAM-only and PCM-only.

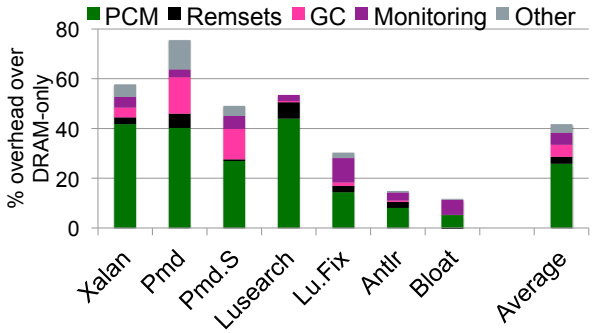


Figure 5.9: PCM access time overheads dominate collector and monitor overheads in KG-W.

Energy

To quantify the energy efficiency of KG-W, KG-N, and a PCM-only system, Figure 5.8 shows the energy-delay product (EDP) normalized to a DRAM-only system. EDP is energy multiplied by execution time, so it takes the higher latencies of PCM into account. The EDP is sometimes worse on a PCM-only system compared to DRAM-only, particularly for lusearch. Using KG-N reduces the average energy-delay product by 36% over the DRAM-only system, and also significantly improves over a PCM-only system. Because of KG-W's additional overhead, its energy-delay product is slightly higher, saving 32% over a DRAM-only system. In addition to reducing the EDP, both KG-N and KGW reduce the total energy consumption by 47% on average.

Breakdown of Overheads

A PCM-only system adds 70% to the execution time of a DRAM-only system on average. Our simulator results show that KG-N reduces overheads by over 50% compared to PCM-only, but still adds 31% to execution time on average over DRAM-only. KG-W adds overhead compared to KG-N (40% on average) because it monitors individual objects and copies long-lived objects at least one more time than KG-N, since it first copies them to the observer space and then to a mature space.

We break down KG-W overheads into: (a) Remsets — write-barriers to remember pointers to the observer space; (b) GC — KG-W adds collections of the observer space, which often adds overhead; however, collection time sometimes reduces because objects die in the observer space, reducing full heap collections; (c) Monitoring — KG-W's write-barrier records more information about all writes to non-nursery objects; and (d) PCM — PCM has longer read and write latencies than DRAM. Other effects, such as cache locality, are unmeasured in this experiment and we report them in (e) Other. We configure the simulator and the VM in a variety of different ways to measure all these overheads.

Figure 5.9 presents this breakdown relative to DRAM-only. The largest overhead is the longer PCM access latencies (PCM), which add 25% to total time on average. The overhead of collecting KG-W's extra spaces (GC), and of monitoring writes to non-nursery spaces (Monitoring) are each a little under 5% on average. Keeping track of more remembered sets to collect the observer space in isolation (Remsets) adds around 3% overhead. Other overhead (Other) accounts for another 3% of extra execution time. Pmd has a high Other overhead because it has a high nursery survival rate (see Table 5.4) which triggers more observer collections and has cache effects. Our real system performance results in Section 5.6.2 confirm that the Kingsguard mechanisms themselves add little to total time. Using a hybrid DRAM-PCM memory system for its scalability properties inevitably adds latency to execution times. KG-W mitigates these latencies by redirecting some reads and writes to a small amount of DRAM.

The Origin of Writes

Figure 5.10 classifies where writes to PCM originate: the application, nursery collection, observer collection, or major collection. We modify the simulator to track which phase last wrote each cache line, since LRU policies evict lines to PCM or DRAM well after their last access. KG-W reduces PCM application writes for most benchmarks compared to KG-N. This reduction corresponds to an increase in DRAM writes (not shown), as designed. The average increase in writes to both DRAM and PCM together (not shown) with KG-W over KG-N is 12% and the worst case is pmd at 25%. This increase stems from additional collection work. Figure 5.10 shows only the writes to PCM.

The collector induces writes when it initially places an object in PCM and when it updates PCM references to other objects. When analyzing the writes performed by the collector, we note that: (1) KG-N incurs writes to PCM during a nursery collection both due to copying survivors into the PCM mature space *and* due to updating the references in PCM that point to them; (2) KG-W eliminates major collections for lu.fix and bloat by reclaiming objects in the observer space; (3) writes to PCM during a nursery collection with KG-W are solely due to updating references in PCM spaces that point to surviving nursery objects copied to the observer space. These results suggest further PCM write reductions are possible by avoiding pointer updates in PCM and deploying better predictors of application writes.

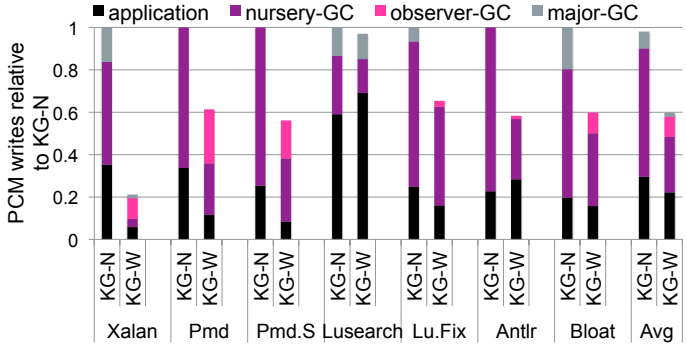


Figure 5.10: Where writes to PCM originate. KG-W reduces application and collector writes to PCM.

5.6.2 Real Hardware Results

We now evaluate Kingsguard on real hardware. We analyze write behavior, performance, and memory characteristics. Lacking PCM hardware, all latencies are to DRAM.

Write Analysis

Figure 5.11 presents writes to the PCM heap using KG-N and KG-W as reported by write-barriers. These results are thus architecture-independent since they do not consider cache effects that filter out some writes to both DRAM and PCM. We normalize to KG-N with a 4 MB nursery and compare to KG-N with a larger 12 MB nursery, and to KG-W with a 4 MB nursery with and without primitive write-barriers. Using a larger nursery reduces the writes to PCM by 24% on average compared to KG-N. A larger nursery is not effective at reducing PCM writes for four out of the five applications with more writes in the mature space than the nursery (the five left-most applications in this figure and in Figure 5.2). KG-W is much more effective than simply using a larger nursery, reducing writes to PCM by 80% on average.

For sunflow, KG-W eliminates 99.7% of writes to PCM by copying the written objects to mature DRAM during observer collections. For pjbb and hsqldb, we similarly observe many writes to the few mature DRAM objects. On the other hand, KG-W eliminates 97% of writes to PCM for lusearch by moving primitive arrays from PCM to DRAM during mature collections. KG-W reduces writes for all the GraphChi benchmarks, which all need very large heaps, by over 50% as compared to KG-N. For luindex and CC, large objects in PCM incur a lot of writes, and are only moved to DRAM during a mature collection. Interestingly, luindex with KG-W requires no mature space collections because so many objects die in the observer space. For CC, writes happen before a mature collection is triggered. These behaviors motivate additional policies for mature collection to be triggered by writes to PCM. We leave this exploration to future work.

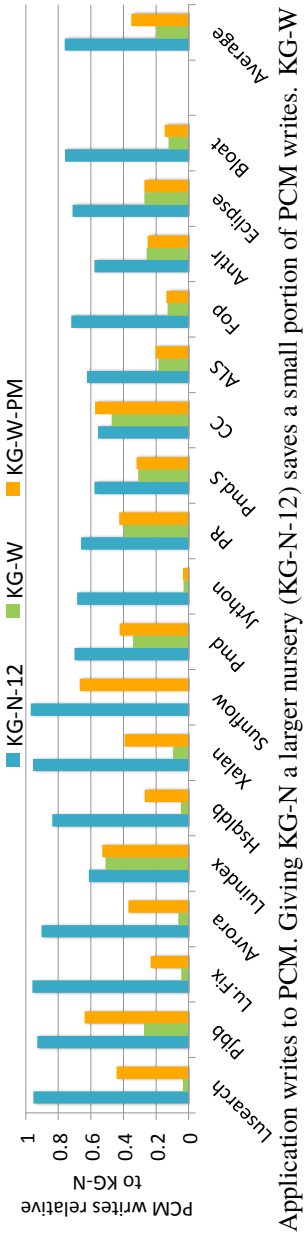


Figure 5.11: Application writes to PCM. Giving KG-N a larger nursery (KG-N-12) saves a small portion of PCM writes. KG-W saves 80% of PCM writes compared to KG-N. Excluding primitive monitoring of writes (KG-W–PM) increases writes to PCM.

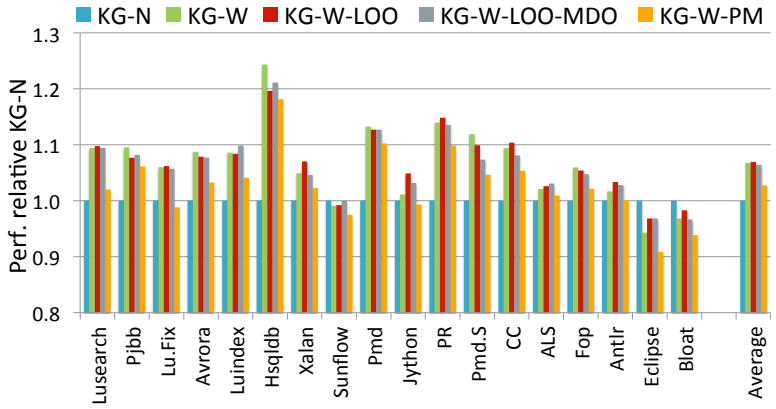


Figure 5.12: KG-N performs best. KG-W adds 7% overhead to execution time on average.

Primitive versus reference monitoring. We observe that excluding primitive monitoring (KG-W–PM) in Figure 5.11 significantly reduces writes compared to KG-N for several applications. For instance, in the case of pmd, eclipse, and bloat, reference writes capture most of the highly mutated objects. On the other hand, for seven applications, PCM writes increase quite a bit over KG-W. On average, KG-W-PM eliminates 65% of PCM writes compared to 80% for KG-W.

Performance

Figure 5.12 presents the performance of KG-W configurations normalized to KG-N. The results in Figure 5.12 understate the performance advantage of KG-W over KG-N. On a system with PCM, the large reduction in writes to PCM reduces execution time due to latency savings. With respect to KG-W, the large object (KG-W–LOO) and metadata space optimizations (KG-W–LOO–MDO) are performance-neutral on average. KG-W increases the execution time on average by 7%, and hsqldb by 25% over KG-N. This overhead is mostly due to the additional observer collections. During each observer collection, some highly mutated objects are placed in mature DRAM, which results in a large reduction in writes to PCM, as shown in Figure 5.11. KG-W reduces execution time for a few benchmarks: sunflow, eclipse, and bloat, due to fewer full-heap collections. This result is a feature of KG-W. Observer collections are cheaper than full-heap collections because (1) they operate over smaller regions and (2) when they reclaim objects, they prevent mature DRAM and PCM from filling up. Finally, we observe that eliminating primitive monitoring (KG-W–PM) has the highest impact on lusearch: a 7% reduction in execution time.

Memory and Demographic Analysis

Table 5.4 shows allocation and survival rates, and heap space occupancy per benchmark for our collectors. Applications allocate frequently, between 56 MB and 14 GB of memory (column 1), especially the GraphChi benchmarks. We gray out those

applications with less than 100 MB of allocation and exclude them from the averages. Our applications have an average nursery survival rate of 17% and a maximum of 66% (columns 3 and 4).

A 4 MB DRAM nursery maximizes the use of PCM for 97% of the KG-N heap, averaging between 21 and 280 MB, and up to 502 MB of PCM, for the GraphChi benchmarks (columns 5 and 6). KG-W uses more DRAM: 6 to 86 MB of DRAM on average and up to 224 MB (columns 9 and 10), and 16 MB to 263 MB of PCM and up to 484 MB for CC (columns 7 and 8). KG-W trades higher utilization of DRAM (10%) for disproportionate increases in PCM liftetimes.

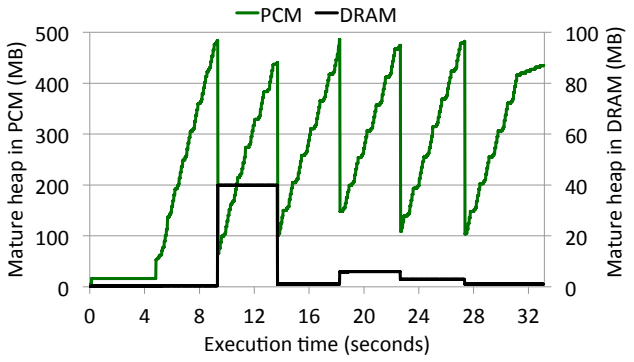
Columns 11 and 12 show the DRAM consumed by WP. The average DRAM consumption is 7% more than KG-W. Individual benchmarks behave differently. For instance, for lusearch and xalan, WP consumes 3× and 5× more DRAM on average. Both these benchmarks allocate many large objects directly in PCM. WP's reactive algorithm keeps these pages in DRAM. For the remaining benchmarks, WP consumes less or similar DRAM memory compared to KG-W.

Column 13 reports the percent of the total heap (column 1) occupied by KG-W's mature DRAM space, which ranges from 1.3 MB to 186 MB, only 8% of the heap on average. Columns 14 and 15 show that the DRAM metadata space consumes a small fraction of the KG-W heap. Overall, KG-W stores 80% of the heap in PCM versus KG-N's 98%.

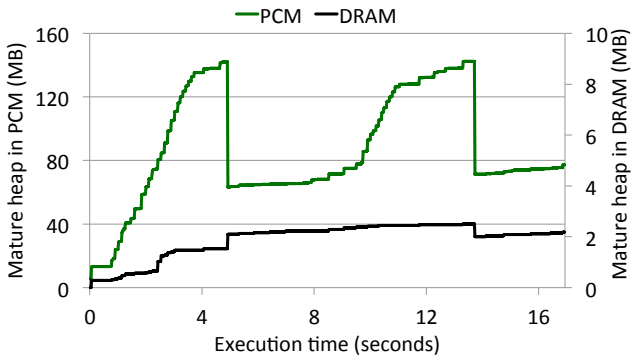
Column 16 shows that the observer-space survival rate ranges from 0.2% to 99%. Benchmarks with large observer survival rates, such as hsqldb, PR, pmd, pmd.S, and CC have correspondingly higher overheads in Figure 5.12 due to having to copy objects twice before they reach the mature space. In contrast, the benchmarks with low observer space survival rates have lower overheads. The last columns shows KG-W copies most objects to PCM: it retains only between 0.2% and 41% of surviving observer objects in DRAM. KG-W uses less PCM memory than KG-N for two reasons: objects die in the observer space and KG-W keeps a few written objects in mature DRAM. Even though our applications exhibit a wide range of behaviors with respect to object demographics and writes, KG-W is extremely effective at limiting PCM writes by managing object placement and migration in hybrid DRAM-PCM memories.

Table 5.4: Object demographics. KG-N maximizes the use of PCM, whereas KG-W more selectively uses PCM since (1) many objects die before promotion and (2) KG-W retains a small fraction of objects in DRAM. Columns 12 and 13 show the DRAM consumed by WP for the simulated benchmarks.

	allocation MB (1)	Heap MB (2)	% nursery survival KG-N (3)	% nursery survival KG-W (4)	KG-N PCM MB avg (5)	KG-N PCM MB max (6)	KG-W PCM MB avg (7)	KG-W PCM MB max (8)	KG-W DRAM MB avg (9)	KG-W DRAM MB max (10)	WP DRAM MB avg (11)	WP DRAM MB max (12)	KG-W % mature in DRAM (13)	KG-W metadata MB avg (14)	KG-W metadata MB max (15)	KG-W % observer survival (16)	KG-W % held in DRAM MB (17)	KG-W % held in DRAM Obj (18)
Lusearch	4294	68	4%	4%	49	64	41	63	6	7	31	38	0%	1.8	2.8	29%	6%	9%
Pjbb	2314	400	20%	20%	280	386	252	310	40	52			5%	16	20	84%	7%	6%
Lu.Fix	848	68	2%	2%	22	24	16	16	15	15	7	8	8%	1.6	1.9	25%	3%	4%
Avrora	64	98	15%	15%	24	25	20	20	15	16			0%	3	4	0%	0%	0%
Luindex	37	44	22%	22%	21	22	21	21	13	13			0%	1	1	0%	0%	0%
Hsqldb	165	254	66%	60%	85	137	70	120	19	21			1%	6	8	88%	0.2%	0.02%
Xalan	980	108	17%	14%	73	104	52	92	18	18	51	60	7%	1.8	1.9	9%	8%	1%
Sunflow	1920	108	2%	2%	31	45	21	22	18	18			15%	2	2	13%	35%	30%
Pmd	364	98	23%	23%	73	94	94	47	20	24	13	25	8%	4.3	7.9	68%	5%	7%
Jython	1150	80	0.001%	0.2%	73	80	68	64	21	26			6%	5	10	12%	30%	25%
PR	6946	512	36%	36%	254	502	263	490	32	60			3%	13	6	99%	0.3%	0.1%
Pmd.S	202	98	27%	27%	55	77	33	40	19	20	17	24	11%	2.6	3.5	47%	6%	9%
CC	5507	512	24%	24%	242	502	240	484	35	75			5%	10	22	97%	2%	1%
ALS	14245	512	9%	10%	254	502	215	430	86	224			25%	4	22	63%	50%	41%
Fop	56	80	20%	20%	26	28	24	24	17	17			8%	3	3	82%	14%	7%
Antlr	246	48	15%	15%	27	36	19	20	15	15	7	9	10%	1.1	1.1	0.16%	17%	5%
Bloat	1246	66	4%	4%	34	46	27	27	16	16	7	27	7%	1.9	2.1	19%	13%	12%
Eclipse	3082	160	15%	14%	114	155	110	145	23	25			5%	6	7	37%	1%	1%
Avg (All)	2900	206	17%	17%	111	184	98	140	26	40			8%	5	8	46%	12%	10%
Heap %					97%	98%	80%	80%	20%	20%			8%	2%	4%			
Avg (Sim)	**1169**	**79**	**13%**	**13%**	**48**	**64**	**34**	**47**	**16**	**16**	**19**	**27**	**4%**	**2.2**	**3.0**	**28%**	**8%**	**7%**
Heap %					**92%**	**94%**	**68%**	**88%**	**32%**	**22%**	**39%**	**36%**	**4%**	**4%**	**5%**			



(a) Page Rank



(b) Eclipse

Figure 5.13: MB in PCM (left-hand y-axis label) versus MB in DRAM (right-hand y-axis label) as a function of time for Page Rank and Eclipse. KG-W uses large amounts of PCM and small amounts of DRAM.

Heap Composition

This section explores the way KG-W uses DRAM and PCM using heap composition graphs. Figure 5.13 plots the usage of PCM versus DRAM in MB over time for PR and eclipse. For both applications, full heap collections cause the amount of PCM memory used to decrease drastically, mostly because many objects die, but also because some are copied to DRAM.

For PR, while the amount of PCM used grows to close to 500 MB, the maximum amount of DRAM used (right axis) is around 40 MB. Visually, DRAM occupancy increases at the same time as PCM shrinks due to a full heap collection that moves objects from PCM to DRAM. For instance at around 10 seconds into execution, DRAM occupancy increases due to a full heap collection that copies written mature objects out of PCM. As Table 5.4 shows, Page Rank has a high observer survival rate, yet KG-W promotes very few of these surviving observer objects to mature DRAM. The graph shows how these observer collections quickly populate PCM and consequently trigger full heap collections.

We observe that eclipse uses up to 145 MB of PCM, but the DRAM usage maxes out around 2.5 MB. Highly mutated objects in DRAM also have long lifetimes for eclipse. During mature collections (around the 5 second and 14 second marks), whereas the PCM usage drops by almost half, much of mature DRAM stays alive. Table 5.4 shows that observer collections copy only 1% of surviving objects on average to mature DRAM. Fortunately, these DRAM objects are highly written, which protects PCM from writes.

5.7 Summary and Interpretation

This chapter introduces write-rationing garbage collectors, which seek to maximize the use of PCM while improving its lifetime in hybrid memory systems. Our Kingsguard collectors exploit object demographics and individual object write behaviors in Java applications. Kingsguard-nursery (KG-N) places nursery objects in DRAM and all other objects in PCM. Kingsguard-writers (KG-W) adds monitoring of mature object writes and moves objects between DRAM and PCM based on their individual write history. KG-N places 92% of heap objects on average in PCM, but still removes over 80% of writes to PCM compared to PCM-only with hardware wear-leveling, leading to a 5× improvement in PCM lifetime. KG-W places 68% of the heap in PCM to remove over 90% of all writes, thus greatly extending PCM lifetime by 11×. Both KG-N and KG-W improve over WP, the state-of-the-art OS approach [105, 131]; WP writes to PCM 3× more than KG-W. Overall, this work demonstrates that managed runtimes have a significant advantage over hardware and OS-only approaches because they can exploit, observe, and react to coarse-grained object demographics and to fine-grained object behaviors, opening up a new and promising direction to manage hybrid DRAM-PCM memory systems.

Researchers are still developing non-volatile memory technologies, so endurance levels, access latency, and energy characteristics may improve. Regardless, PCM cell endurance is very unlikely to reach DRAM levels because of material properties. Other technologies, such as resistive random-access memory prototypes, have higher, but still finite endurance, e.g., one trillion writes per cell [83]. We believe write-rationing garbage collection is also relevant for other memory technologies.

Unfortunately, KG-W suffers from three main drawbacks. (1) It incurs high overhead from monitoring objects to dynamically discover frequently written objects. (2) It is reactive; it monitors object writes in a limited time window and must wait until the next collection to act on the information, leading to mispredictions and allocation of frequently written objects in PCM, particularly large objects. (3) It consumes excessive DRAM capacity. The next chapter introduces a different approach towards write-rationing garbage collection that addresses the drawbacks of KG-W.

Chapter 6

Crystal Gazer: Profile-Driven Write-Rationing Garbage Collection for Hybrid Memories

6.1 Introduction

Including PCM into the main memory system requires solutions to tolerate the limited write endurance. The previous chapter introduced *write-rationing garbage collection* for hybrid DRAM-PCM memory to improve endurance by placing frequently written objects in DRAM spaces and read-mostly objects in a PCM mature space [2]. The best write-rationing collector, Kingsguard-Writers (KG-W), has shortcomings: it is reactive and suffers from high overhead, and consumes excessive DRAM capacity.

This chapter introduces profile-driven write-rationing garbage collection for hybrid memories, called Crystal Gazer (CGZ). We leverage the fact that modern mobile and server workloads execute frequently, which makes profiling practical. Prior work shows that allocation site, or the code location where the object is allocated, is a good predictor of object lifetimes [8, 19, 15, 29, 68], and we find it is also a good predictor of write-intensity. We first profile individual object writes and their allocation sites, classifying a site as producing read-mostly or highly written objects in an offline run. We show that the predictor is highly accurate with *true advice* (different inputs for training than classification) for 15 Java benchmarks from three suites (DaCapo, Pjbb and GraphChi).

CGZ uses the profile at runtime to guide object placement in mature DRAM-backed and PCM-backed memory spaces. CGZ initially allocates all objects in the DRAM nursery. It uses the advice to label objects at allocation time as coming from read-mostly or highly written allocation sites. When it promotes a nursery survivor, it copies the object to the DRAM or PCM mature space according to the predicted write-intensity label. If there is no advice at all, a production system should fall back on KG-W for dynamic monitoring. Unprofiled allocation sites may default to PCM or

DRAM. CGZ promotes frequently written objects to DRAM, protecting PCM from writes. It promotes read-mostly objects to PCM, exploiting PCM capacity. It places the majority of mature objects in PCM, because only a small fraction of objects are frequently written.

Leveraging profile information overcomes the three major drawbacks of existing write-rationing garbage collectors. (1) Profile-driven promotion to mature DRAM and PCM spaces eliminates the overhead of dynamically monitoring objects. (2) CGZ is proactive in placing objects in DRAM or PCM, which combined with the high accuracy of ahead-of-time profiling, reduces mispredictions, particularly of large objects, and the need to copy objects between spaces, further extending PCM's lifetime. (3) Because writes are concentrated in a small fraction of objects and well predicted by allocation site, it reduces the amount of DRAM capacity needed, leveraging PCM's large capacity.

A key feature of CGZ is its ability to trade off PCM lifetime for DRAM capacity by using different heuristics and thresholds to classify allocation sites, using only one profiling run. Our experimental evaluation shows that CGZ provides a Pareto-optimal tradeoff between PCM lifetime and DRAM capacity. In contrast, KG-W provides a single sub-optimal operating point. Our experimental evaluation with 15 Java workloads uses an emulated hybrid memory system on multi-socket NUMA hardware to explore CGZ's effectiveness. We bind the application to one socket and emulate DRAM as the local NUMA node memory and PCM as the remote NUMA node memory. Compared to KG-W, the state-of-the-art in terms of improving PCM lifetime for hybrid memories, CGZ reduces the execution time overhead by 8% on average and up to 30%. CGZ also eliminates 30% more PCM writes on average than KG-W, when optimized for extending PCM's lifetime. It consumes 68% less DRAM capacity, when optimized for the smallest DRAM capacity.

Counter-intuitively perhaps, the *static* profile-driven CGZ solution outperforms KG-W's *dynamic* approach. The reason is twofold. (1) Allocation site is a good predictor for write-intensity, i.e., most objects allocated from a single site are either frequently written or read-mostly — we refer to this property as allocation site write homogeneity. (2) A small number of allocation sites captures the bulk of writes to a small fraction of the entire mature space heap volume. The high prediction accuracy for write-intensive objects allocated from a limited number of allocation sites makes CGZ outperform KG-W, which needs to dynamically learn object write-intensity and may place highly written objects in PCM, which it cannot move to DRAM until the next full-heap collection.

Overall, this chapter makes the following contributions:

- the design and implementation of profile-driven write-rationing garbage collection for hybrid memories;
- offline profiling, advice generation, and a compilation framework that gathers, generates, and uses allocation advice to trade off PCM lifetime and DRAM capacity;
- emulation results demonstrating reduced execution time overhead compared to state-of-the-art write-rationing garbage collection while at the same time

extending PCM lifetime and reducing DRAM capacity needs.

6.2 Related Work and Background

Chapter 2 provides background on our assumed memory and storage hierarchy, Java Virtual Machine (JVM), garbage collection, and the specific Immix garbage collector on which we build. This section discusses related work on profile-based optimizations for Java workloads.

Profile-driven DRAM memory management. Prior works use allocation site profiling to optimize the performance and energy of DRAM-based memory systems. Jantz et al. [63] use offline profiling to divide allocation sites into hot and cold sites. The heap is partitioned into hot and cold regions, and the OS maps virtual to physical pages with the goal to reduce DRAM energy consumption without hurting performance. In contrast, we focus on memory lifetimes in hybrid DRAM-PCM memories. Our work is the first to show the allocation site homogeneity of writes, and a garbage collector that acts upon allocation site advice to place highly written objects in DRAM. Recent work also exploits allocation site profiling to place program data in upcoming memory systems with high-bandwidth DRAM placed next to traditional DRAM [41]. Their work is limited by C++ semantics. In contrast to our work, they do not use a real-world hardware prototype to evaluate their data placement strategies and hence, among other limitations, are unable to report the total execution time of their applications.

Profile-driven pretenuring. Prior work profiles allocation sites to predict object lifetimes and allocate long-lived objects directly in a mature space (pretenuring), eliminating nursery promotion costs. We follow the same approach of ahead-of-time profiling and then applying advice in production runs as in Blackburn et al. [19, 15], but based on writes instead of lifetime. Dynamic lifetime profiling has the advantages that it does not require a profile and can react to program phases, but the disadvantages of dynamic monitoring costs and warm up time [68]. In our work, advice predicts write-intensity and the collector allocates objects in DRAM or PCM, both at allocation (large objects) and promotion (small objects) time. We find allocation site better predicts write-intensity than lifetime and believe combining both predictions, and some dynamic monitoring, are interesting avenues for future work.

Other profile-driven optimizations. Krintz et al. [74] profile Java applications offline to discover compiler optimizations that speed up a method's execution. They annotate the bytecode to communicate these optimizations to the compiler which reduces compilation overhead during run-time. In subsequent work, Krintz [73] combines offline and online profiling to reduce compilation overhead even further. Profiling has been used to improve memory management in Java workloads. Buytaert et al. [23] collect information offline about when to trigger garbage collection to maximize collection yield. They also use offline analysis to decide, during execution

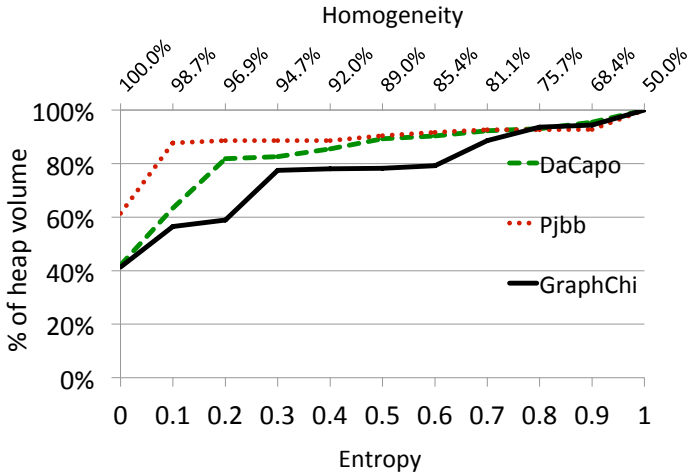


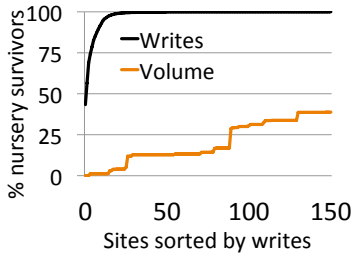
Figure 6.1: Write homogeneity for allocation site. *40-60% of the heap volume is allocated from a perfectly homogeneous allocation site; 80-90% of the heap volume is allocated from allocation sites that are at least 90% homogeneous.*

time, between triggering nursery or full-heap collections. Chen et al. [28] leverage profile information to proactively reorganize the heap to improve data locality.

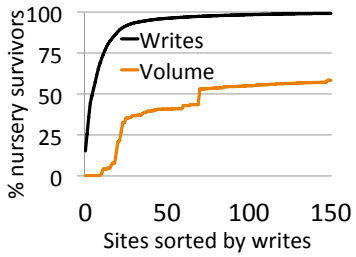
6.3 Allocation Site as a Write Predictor

This section examines how well allocation site predicts writes to objects and the distribution of writes in heap memory. We use 15 Java workloads, including modern transaction and graph processing workloads with huge memory footprints. Prior work establishes allocation site as an accurate predictor of object lifetime [8, 19, 15, 29, 68]. Our prior work shows that nursery objects incur many writes and thus, to avoid writes to NVM, should be put in DRAM [2]. We show here that allocation site is also a good predictor of the write-intensity of old objects and that these writes are concentrated to a small volume of objects.

Allocation site homogeneity. To assess the predictive power of allocation site for object write-intensity, we measure the homogeneity of writes on the basis of allocation site. We use the information-theoretic notion of entropy to capture write homogeneity. An entropy of 0 means perfect homogeneity, i.e., 100% of objects are highly written or read-mostly. A homogeneity of 1 means no homogeneity, i.e., 50% of objects are highly written and 50% are read-mostly. To compute entropy, we classify objects as highly written if they are written once after allocation in the mature space, and read-mostly otherwise. Figure 6.1 shows average homogeneity curves for the DaCapo, Pjbb and GraphChi benchmark suites. The percentage of heap volume is reported as a function of allocation site homogeneity. The bottom and top horizontal axes report entropy and the fraction of objects that are homogeneous in write-intensity, respectively. The homogeneity of allocation sites is very high: 40% to 60% of the



(a) Pjbb



(b) Page Rank

Figure 6.2: Distribution of mature writes and heap volume by allocation site for Pjbb and Page Rank. *A few sites capture a majority of mature object writes and occupy a small fraction of the heap.*

heap volume is perfectly homogeneous, and 80% to 90% is from sites with very high (at least 90%) homogeneity.

The observation that allocation site is a good predictor of write-intensity motivates a profile-driven approach, i.e., classifying allocation sites through profiling provides a prediction for object write-intensity. However, allocation site write-homogeneity is not enough for a well-performing write-rationing garbage collector. They also are most efficient if the heap volume of write-intensive objects is small, so that we can allocate the least possible volume in DRAM to leverage PCM's capacity to the fullest.

Write distribution. Figure 6.2 shows the cumulative distribution of writes to objects in the mature space and its heap volume as a percentage of the total mature allocation on a per allocation site basis for two representative benchmarks: Pjbb (most homogeneous) and GraphChi's Page Rank (least). We observe that a couple dozen sites out of a couple thousand capture the vast majority of mature writes and constitute only a small fraction of the total heap volume. Similar results hold for all other benchmarks. These two key observations reveal the opportunity that we exploit in Crystal Gazer: profiling accurately identifies sites that allocate a small volume of highly written objects.

6.4 Crystal Gazer

This section describes Crystal Gazer, profile-driven write-rationing garbage collection for hybrid memories. CGZ uses offline profiling to analyze object write-

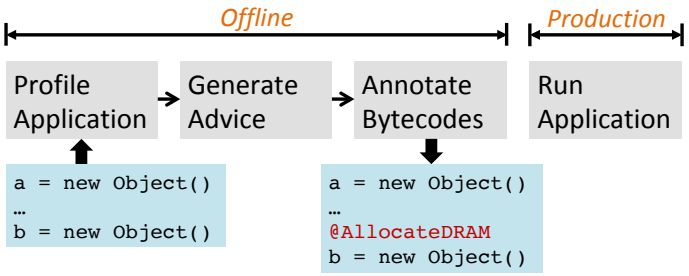


Figure 6.3: Overview of Crystal Gazer. Offline analysis identifies allocation sites of highly written objects (e.g., object b) which are annotated in the bytecode. During production, the collector will allocate an object in DRAM if predicted highly written versus PCM if predicted read-mostly, upon a nursery collection.

intensity, from which we generate advice to be used in a subsequent production run. CGZ eliminates the high cost of dynamic monitoring and unnecessary copying compared to KG-W, the previous best write-rationing garbage collector. CGZ achieves a combination of better performance, reduced DRAM usage, and fewer writes to PCM than KG-W.

6.4.1 Overview

Figure 6.3 shows the Crystal Gazer work flow. We first profile the application to collect a trace of writes to each object and their allocation sites. We group objects by allocation site and use various heuristics to label allocation sites as *DRAM* (i.e., objects allocated from this site are predicted frequently written) or *PCM* (i.e., objects allocated from this site are predicted read-mostly). Advice files record allocation sites labeled *DRAM*. All other sites are implicitly labeled as *PCM*. Thus, an unprofiled site may be labeled either DRAM or PCM; we default to PCM in this work. For expediency, we use bytecode rewriting to insert a new_dram bytecode based on the profile. The standard portable mechanism is to annotate bytecodes [74], since Java compilers simply ignore unsupported annotations. During a production run, our modified compiler generates a special allocation sequence to process the new_dram bytecode that, in addition to reserving the space, labels objects as DRAM or PCM. Crystal Gazer then promotes objects from DRAM-labeled sites to a mature DRAM space and other objects (expected to be read-mostly) to a mature PCM space.

6.4.2 Profiling

Our prior work shows that nursery objects plus a small fraction (2%) of all mature objects capture 90% of all application writes [2]. The KG-W write-rationing garbage collector incurs significant overhead to discover the highly written 2% of mature objects. Our offline profiling eliminates this overhead by identifying allocation sites that produce highly written objects in previous executions. Profiling produces a write-intensity trace that records for each object: (1) a unique identifier, (2) the number of writes, (3) its size in bytes, and (4) the allocation site. See Figure 6.4(a) for an example.

The last column lists the object's allocation site as a <method-name:bytecode-index> pair. Because most objects die young, we only profile mature objects which also reduces the size of the write-intensity trace.

To identify objects, we use mature space addresses. We configure the heap size to use the entire 32-bit virtual address space in Jikes RVM. This setting eliminates full-heap collections for DaCapo benchmarks and SPECjbb. As a result, each object has a unique address in the trace. The GraphChi benchmarks allocate more memory and thus require full-heap collections. In this case, we compute the write-intensity trace per full-heap collection. We process individual traces and combine them to gather allocation advice for the entire application. An alternative option would be to consolidate object statistics on the fly upon full-heap collections.

We use the same nursery size during profiling as we use during a production run. If the production nursery size is unknown, a small (thus conservative) nursery will capture a large fraction of mature objects and their allocation sites in the write-intensity trace. Using small nurseries during profiling produces write-intensity characteristics for more objects, but increases the size of the write-intensity trace.

We use write barriers to count the number of writes to each object. Reference write barriers are required for all generational collectors to collect the nursery independently, to record old to young pointers. We also enable write barriers to primitives during profiling. Profiling ignores zero-initializing writes. Write barriers capture all writes regardless of whether the object or any of its fields are physically in a processor cache or main memory. Our profiling is thus architecture-independent, as is the allocation advice we produce for Crystal Gazer.

During profiling, we label objects with their allocation site at allocation time. We associate each allocation site with a unique identifier which the compiler creates when it first encounters each new bytecode during profiling, following prior work [60]. The compiler generates an allocation sequence that stores this identifier in the header of each object. At the end of program execution, we record the allocation site along with each object's address and other attributes in the write-intensity trace. To correlate allocation sites between executions of an application, the trace records the class, method, and bytecode index of the allocation site.

Collecting a write-intensity trace incurs a 2.4× slowdown over native execution on average according to our measurements. The trace's size ranges between 200 KB and 120 MB after compression for our benchmarks. We did not optimize this overhead further since it is incurred infrequently in non-production runs.

6.4.3 Allocation Site Classification

Next, we analyze the write-intensity trace to generate allocation advice, classifying allocation sites as *DRAM* versus *PCM*. We use two criteria for classification. (1) The fraction of total objects allocated from a site that are write-intensive. (2) Thresholds that define write-intensive objects. For the first criterion, we use a *write homogeneity* threshold. If the fraction of write-intensive objects allocated from a site is above the write homogeneity threshold (θ_h), we classify the site as *DRAM*. Otherwise, we classify the site as *PCM*. A small homogeneity threshold works best to limit the number

Object	Writes	Bytes	Method:Idx
O1	0	4	A():10
O2	0	4	A():10
O3	1	4	A():10
O4	1	4	A():10
O5	16	16	A():10
O6	1024	4096	B():4

(a) Example write intensity trace

Heuristic	θ_h	θ_f	θ_d	DRAM Sites
FREQ	5%	1	X	A & B
FREQ	5%	10	X	A & B
FREQ	5%	100	X	B
DENS	5%	X	0.1	A & B
DENS	5%	X	1	A
DENS	5%	X	10	None

(b) Allocation site prediction

Figure 6.4: Example of a write-intensity trace with allocation sites in the last column (a) and prediction of allocation sites using the FREQ and DENS heuristics (b).

of writes to PCM but puts more pressure on DRAM capacity. A high homogeneity threshold reduces DRAM capacity usage at the expense of more PCM writes. For the second criterion that classifies an object as write-intensive, we consider two heuristics.

Write-Frequency (FREQ) uses the frequency of writes to identify write-intensive objects. If an object gets more than a write-frequency threshold θ_f of writes, the object is considered write-intensive.

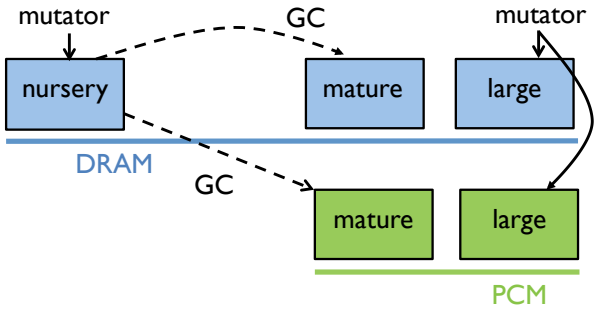
Write-Density (DENS) uses the ratio of writes to object size (in bytes) to identify write-intensive objects. Objects with a write-density above a write-density threshold θ_d are considered write-intensive. DENS gives higher weight to small objects that collect a relatively large number of writes. DENS prioritizes small objects for DRAM allocation and large objects for PCM allocation, thereby better exploiting PCM's capacity compared to FREQ.

Example. Figure 6.4(a) shows an example write-intensity trace consisting of 6 objects from two allocation sites: from method A and B. We analyze the trace using the FREQ and DENS heuristics, and identify which of the two sites are classified as *DRAM* in Figure 6.4(b). We assume a homogeneity threshold of 5%. We increase θ_f from 1 to 100, and θ_d from 0.1 to 10 to observe their impact on site classification. Setting θ_f to 1 classifies both A and B as *DRAM*. Raising θ_f to 100 excludes A from *DRAM* classification because it does not have 5% of objects with more than 100 writes. However, B will still be labeled as *DRAM*. If we want to preserve DRAM capacity by excluding B, we need even larger values for θ_f. On the other hand, if we consider DENS, which uses less DRAM capacity, both A and B are classified as *DRAM* with a θ_d of 0.1. When we increase θ_d to one, only A is classified as *DRAM*. With this setting, objects allocated from B do not get sufficient writes per byte to be classified as *DRAM*. Finally, when θ_d is set to 10, both A and B are excluded from *DRAM* labeling. This example illustrates that large write-density thresholds favor exploiting PCM capacity.

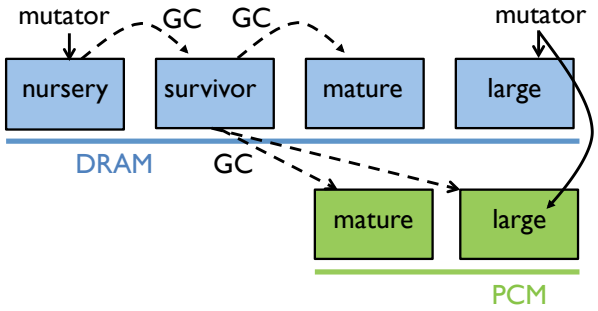
6.4.4 Bytecode Generation

The previous step generates allocation site advice as a file of <site-string, advice> pairs. The advice file only includes the allocation sites labeled *DRAM*. Unlabeled allocation sites default to *PCM*. Future systems could consider labeling unprofiled sites as *DRAM* or dynamically profiling just these objects. Since a minority of allocation sites are labeled *DRAM*, the size of the advice file is minimized.

We use bytecode rewriting to communicate allocation site labels to the managed runtime. The bytecode rewriter first identifies the allocation site, and then queries the advice file to check whether the site is present. If it is not, the rewriter leaves the new bytecode unchanged. If it is, the rewriter overwrites the new bytecode with the newly introduced new_dram bytecode. The runtime, when interpreting or compiling the new bytecode, uses the default allocator, called ALLOC_DEFAULT. The runtime will then copy all objects allocated by such sites to PCM if they survive a nursery collection. For the new_dram bytecode, the runtime uses the newly added ALLOC_DRAM allocator.



(a) Crystal Gazer's heap with no survivor space (CGZ)



(b) Crystal Gazer's heap with survivor space (CGZ-S)

Figure 6.5: Heap organizations without and with a survivor space. Objects are allocated in a DRAM nursery and survivor spaces before being promoted to the mature spaces in DRAM and PCM depending on the object's predicted write behavior.

This allocator sets a bit in the object header which notifies the garbage collector to copy these objects to DRAM if they survive a nursery collection.

6.4.5 Heap Organization

This section describes our heap organizations and how Crystal Gazer copies and allocates highly written objects in DRAM and read-mostly objects in PCM. We consider two heap organizations, see Figure 6.5. They are patterned after Kingsguard heap configurations (for KG-N and KG-W, respectively) to compare apples-to-apples with them. Crystal Gazer collectors follow Kingsguard by always placing new objects in a DRAM nursery, because nursery objects are highly mutated. Some large objects, discussed below, are allocated directly in the mature space. We partition the mature and large object spaces into DRAM and PCM regions. We first describe our system using the heap organization in Figure 6.5(a), and then motivate and describe the heap organization in Figure 6.5(b).

Fresh allocation is a two-step process: (1) reserving space and (2) initializing the object header, called post-allocation. Objects less than 8 KB are always allocated in the nursery. For nursery objects, post-allocation sets a bit in the object's header if its allocation site is labeled DRAM, as shown in Figure 6.6. We steal a bit not in use from the object header in Jikes RVM and call it the DRAM_BIT. Objects with the

DRAM_BIT set are predicted to be highly written. During a nursery collection, the garbage collector checks the DRAM_BIT of each object. If the bit is set, it promotes the object to the mature space in DRAM. Otherwise, it promotes the object, predicted to be read-mostly, to the PCM mature space.

```
1 @Inline
2 public Address postAlloc(ObjectReference ref, int allocator) {
3   if (allocator == Gen.ALLOC_DRAM) {
4     byte old = readHeaderByte(ref);
5     writeHeaderByte(ref, (byte) (old | DRAM_BIT));
6   }
7 }
```

Figure 6.6: Our post allocation sequence sets a special bit in the header of objects that are predicted highly written.

In the default GenImmix, objects larger than 8 KB are allocated directly into a large object space. For these objects, Crystal Gazer's (1) ALLOC_DEFAULT allocates the object directly in the LOS PCM space, and (2) ALLOC_DRAM places the object directly in the LOS DRAM space, as depicted in Figure 6.4(a). Crystal Gazer by default uses both KG-W's metadata optimization and large object optimization (LOO). See Section 6.2. With LOO, the allocator places large objects that are less than half of the remaining nursery size in the nursery to give them a chance to die, because surprisingly some do die quickly. In this case, the object's DRAM_BIT is set based on the advice, and then consulted during the next minor garbage collection to promote the object to the large object space (LOS) in DRAM or PCM.

Copying nursery survivors directly to the mature space results in tenured garbage because some objects die quickly in the mature space. Figure 6.5(b) shows an alternative heap organization with an intermediate space called the *survivor space* between the nursery and the mature spaces. The HotSpot generational collectors use an *eden* space for nursery survivors that serves a similar purpose [38]. The KG-W collector differs because it uses its observer space to monitor writes to these objects as well. In Crystal Gazer, the survivor space's only purpose is to give objects longer time to die and thus limit the amount of tenured garbage. In Crystal Gazer configurations with the survivor space, which we call CGZ-S, nursery survivors are first copied to the survivor space and objects predicted to be highly written carry their DRAM_BIT with them to this space. Next, upon a survivor space collection, the garbage collector checks the DRAM_BIT and copies objects to the mature spaces in DRAM and PCM, accordingly.

6.5 Emulation on NUMA Hardware

There exists no commercially available platform with a hybrid DRAM-PCM memory system. Chapter 5 uses simulation to evaluate the Kingsguard collectors. We first used simulation to evaluate Kingsguard collectors to make it easy to compare to existing OS solution and to reason the energy efficiency of hybrid memory system. Because simulation is time-consuming, it limits the software configurations we can

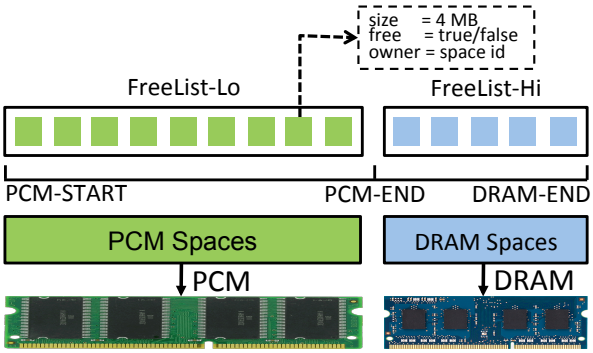


Figure 6.7: The organization of our heap in hybrid memory. Memory composition is exposed to the language runtime. Two free lists keep track of available virtual pages in DRAM and PCM.

evaluate, and precludes critical workloads. In this chapter, we use a hybrid memory emulator to evaluate Crystal Gazer which has the advantage of being fast and flexible.

This section describes the design and implementation of our hybrid memory emulator for managed languages. We discuss the hardware we require to emulate hybrid memory. We present our heap layout in hybrid memory and how we allocate the virtual heap regions in DRAM and PCM. We then provide details for mapping virtual to physical DRAM and PCM memory, and thread scheduling.

6.5.1 Hardware

We use a commodity NUMA platform with two sockets to emulate hybrid memory. We populate both the sockets with DRAM chips. Threads run on one socket, referred to as the local DRAM socket. No threads execute on the other remote PCM socket. Figure 6.8 shows our NUMA hardware platform. Allocation on Socket #0 (S0) is local to the threads and we use it to allocate DRAM memory. Memory accesses on Socket #1 (S1) are remote and emulate PCM.

6.5.2 Heap Layout and Management

The widely used Java runtime environments today manage heap memory using a multi-level hierarchy of blocks and spaces. A space is a coarse-grained partition of the heap. Typically, objects that reside in the same space share a common property. For instance, in generational heaps, the nursery space is used to allocate all the newly created objects. During a (minor) garbage collection, all objects that survive a nursery collection are copied to the mature space. A space is further logically divided into blocks or chunks. The size of the block is a multiple of the page size and it is the minimum unit of virtual memory handed out to a space. A free-list records the location and size of free blocks, and the space to which each block is mapped. The heap manager is responsible for requesting that the operating system maps blocks to physical memory (pages).

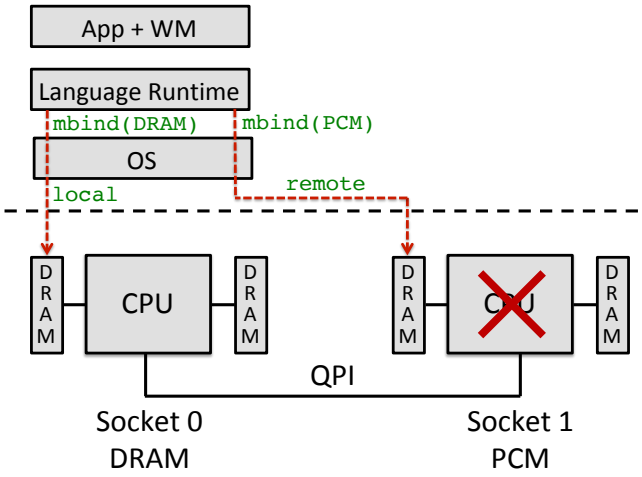


Figure 6.8: Our platform for hybrid memory emulation. The application and write rate monitor (WM) run on Socket #0. The memory on Socket #0 is DRAM and Socket #1 is PCM.

Figure 6.7 provides a high-level view of our heap layout in a hybrid DRAM-PCM system. We use the 32-bit Jikes RVM, but our approach generalizes to other JVMs. In a 32-bit environment, the Linux OS owns the upper 1 GB of the 4 GB virtual memory available to a process. In addition, system libraries use *some* amount of virtual memory for the *malloc* heap. We use the middle 2 GB for the managed heap. We divide virtual heap memory into two portions: (1) a DRAM-backed portion, and (2) a PCM-backed portion. We use a free-list to manage the blocks that belong to each portion: FreeList-Hi and FreeList-Lo. In our heap layout, PCM_START marks the beginning of the user heap, and PCM_END is the end of the PCM-backed portion of the heap, and the beginning of the DRAM-backed portion.

Each space requests that the allocator associated with FreeList-Lo or FreeList-Hi reserve virtual memory. Jikes RVM uses mmap() for reserving virtual memory if none is available as indicated by the free lists. The allocator finds a free chunk and returns the address to the requesting space. The space then makes sure the chunk is mapped in physical memory. In our approach, once a chunk is mapped in physical memory, we do not remove its mapping in the OS page tables even if the chunk is no longer in use by the requesting space. The chunk is recycled by the allocator when another space requests a free chunk. We modify the chunk allocator to map memory in DRAM or PCM.

We use the default size for each chunk in Jikes RVM, i.e., 4 MB. Each entry in the free-list contains meta-information about the chunk: (1) the size, (2) the status (free or in use), and (3) the current owner.

The runtime reserves the address range of the nursery space at boot-time. Similar to the baseline design, we place the nursery at one end of virtual memory. This configuration enables the standard fast boundary write-barrier for generational collection. Other contiguous spaces (such as the observer space in KG-W) are placed next to the nursery. Mature spaces use a request mechanism to acquire chunks at runtime. These

	KG-N		KG-W		CGZ		CGZ-S	
	S0	S1	S0	S1	S0	S1	S0	S1
Nursery	✔	✗	✔	✗	✔	✗	✔	✗
Observer	✗	✗	✔	✗	✔	✗	✔	✗
Mature	✗	✔	✔	✔	✔	✔	✔	✔
Large	✗	✔	✔	✔	✔	✔	✔	✔
Metadata	✗	✔	✔	✔	✗	✔	✔	✔

Table 6.1: Spaces in Kingsguard and Crystal Gazer collectors and their mapping to Socket S0 (DRAM) or Socket S1 (PCM). KG-N does not use an observer space. KG-W uses a mature, large, and metadata space in both DRAM and PCM.

spaces share the pool of available chunks with other spaces. A space is specified as DRAM or PCM using a flag in the constructor of each space.

We allocate memory using the Linux OS calls for specifying a memory allocation on the local or remote memory socket on a NUMA machine. We use the local socket as the DRAM socket and the remote socket as the PCM socket. To bind a virtual memory range to a particular socket, we call mbind() with the socket number after each call to mmap(). We use a NUMA-specific version of the C memory allocator to call these routines. We modify the Java Virtual Machine to call the C routines for DRAM and PCM allocation.

The alternative approach to manage DRAM and PCM spaces is to use a monolithic heap with a single free-list. The efficiency of such an approach is low because it requires unmapping freed chunks from physical memory. If not, a DRAM space could end up using a logical chunk that is physically mapped in PCM. The flexibility of leaving the free chunks mapped in physical memory is a result of our design with two free lists.

6.5.3 Space to Socket Mapping

Table 6.1 shows the space to socket mapping for three of the collectors we evaluate in this work on our emulation platform. KG-W and its variants use extra spaces in DRAM that are mapped to Socket #0 (S0). The observer space in KG-W is placed in DRAM and is used to monitor object writes. KG-W has a mature, large, and metadata space in both DRAM (S0) and PCM (S1). KG-W-MDO does not include the metadata optimization (see Section 6.2). Therefore, it does not use an extra metadata space in DRAM.

The boot space contains the boot image runner that boots Jikes RVM and loads its image files. Except for a system with only PCM, we always place the boot image in DRAM because we observe a large number of writes to it.

6.5.4 Thread to Socket mapping

For the Kingsguard configurations, we always bind threads, including application and JVM service threads, to Socket #0 (see Figure 6.8). When emulating a system with only PCM, we bind threads to Socket #1 for accurately reporting write rates. We do not pin threads to specific cores and use the default OS scheduler.

We measure write rates on our emulation platform using a write rate monitor (WM in Figure 6.8) that also runs on Socket #0. We experimentally find out that scheduling WM on Socket #0 leads to more deterministic write measurements.

6.6 Experimental Methodology

This section discusses experimental methodology including the experimental platform, workloads, and the different write-rationing garbage collector configurations that we evaluate.

Hardware Platform. We use a two socket Intel Sandy Bridge E5-2650L processor. Each socket has 8 physical cores and two hyperthreads per core. The platform features 132 GB of main memory, evenly distributed between the two sockets. We use all DRAM channels on both sockets. All cores share the 20 MB LLC on each processor. The maximum bandwidth to memory is 51.2 GB/s, more than the maximum bandwidth consumed by any of our workloads. The two sockets are connected via a QPI link that supports up to 8 GT/s. We use Ubuntu 12.04.2 with a 3.16.0 kernel. We use Intel's pcm-memory utility from the Performance Counter Monitor framework for measuring write rates.

Measurement Methodology. We use best practices from prior work for evaluating Java applications [51, 61], including replay compilation to eliminate non-determinism due to the optimizing compiler. Replay compilation requires two iterations of a Java application in a single experiment. During the first iteration, the VM compiles each method to a pre-determined optimization level recorded in a prior profiling run. We also generate the appropriate allocation sequence (ALLOC_DEFAULT versus ALLOC_DRAM) depending on the site's classification. The second iteration does not recompile methods leading to steady-state behavior. We take our measurements during the second iteration. This run is deterministic and primarily measures the application performance, instead of the adaptive compiler or JVM start-up behavior. We perform each experiment four times and report the arithmetic mean.

Our emulation platform does not accurately represent end-to-end performance of a hybrid memory system because the access latency to remote memory is much less than to PCM. It does, however, accurately captures the execution time overhead of the Crystal Gazer modifications to the Java managed runtime on real hardware. Most importantly for optimizing lifetime, it accurately represents the number of PCM writes as measured by accesses to remote memory. Furthermore, it accurately represents the use of DRAM capacity with local memory usage. Accessing remote memory incurs

a slight performance degradation compared to local memory, which we find to affect performance by approximately 1% on average and up to 6% for Pjbb.[1]

Java Applications. We use 15 Java applications from three diverse sources: 11 from DaCapo [14], pseudojbb2005 (Pjbb) [16], and 3 applications from the GraphChi framework for processing graphs [78]. The GraphChi applications include page rank (PR), connected components (CC) and ALS matrix factorization (ALS). Compared to prior work [2], we drop jython as it does not execute stably with our Jikes configuration. We use an updated version of lusearch, called lu.Fix, that eliminates useless allocation [129]. To match our hardware platform, we run the multithreaded DaCapo applications, Pjbb and GraphChi applications with four application threads. We use the default data sets for profiling with the DaCapo benchmarks; we use 8 warehouses and 10 K transactions for Pjbb; for GraphChi's PR and CC, we process 1 M edges using the LiveJournal online social network [85], and for ALS, we process 1 M ratings from the training set of the Netflix Challenge. Our datasets for production runs differ from profiling runs. For production runs, we use the large data set for DaCapo; 4 warehouses and 50 K transactions for Pjbb; and a different set of randomly chosen 1 M edges and ratings for GraphChi.

Workload Formation. Multiprogrammed workloads better reflect real-world server workloads because: (1) a single application does not always scale with more cores, and (2) servers typically execute multiple programs to amortize costs. Our multiprogrammed workloads consist of four instances of the same application. To avoid non-determinism due to sharing in the OS caches in multiprogrammed workloads, we use independent copies of the same dataset for the different instances. All four application instances in our multiprogrammed workloads synchronize at a barrier and start the second iteration at the same time. We take the execution time of the longest running instance as the total execution time to run the workload.

Nursery and Heap Sizes. Nursery size affects performance, response time, and space efficiency [6, 12, 121, 132]. We use a nursery of 4 MB for DaCapo and Pjbb. Because a 32 MB nursery improves performance over 4 MB for the GraphChi applications, we use a 32 MB nursery for them. We use a modest heap size that is twice the minimum heap size, reflecting typical production heap sizes and prior work [1, 18, 108, 132, 94]. For all the benchmarks, Table 6.2 lists the heap sizes, the total allocation in MB, nursery and survivor space survival rates, and other statistics (discussed later).

Garbage Collectors and Configurations. We compare Crystal Gazer against the state-of-the-art Kingsguard KG-N and KG-W collectors. Because KG-N is the most basic design and straightforward to implement for a hybrid memory system, we normalize to KG-N as our baseline. Prior work demonstrated KG-N's efficiency and huge write reductions compared to a PCM-only system [2]. Similar to prior work, we place the stack, and two smaller heap spaces, boot and meta-data, in DRAM. We set

[1] This overhead was measured by comparing the performance of Crystal Gazer with the entire heap in remote memory versus local memory on our emulation platform.

the observer space in KG-W, and the survivor space in CGZ-S to twice the size of the nursery. We use two garbage collection threads which is best for Immix [40]. We use a homogeneity threshold of 1% as the default, which we find to be a good compromise between PCM lifetime and DRAM capacity. We present four CGZ configurations: CGZ-F1, CGZ-S-F1, CGZ-D1, and CGZ-S-D1 that vary the optimization goal and include/exclude the survivor space. The CGZ configurations with 'S' include the survivor space, the others do not. We use $\theta_f = 1$ for the FREQ heuristic (denoted 'F1') which minimizes the number of PCM writes, and $\theta_d = 1$ for the DENS heuristic (denoted 'D1'), which minimizes the amount of DRAM capacity.

6.7 Results

We compare Crystal Gazer configurations to the Kingsguard collectors discussed in Chapter 5 along three primary metrics: (1) writes to PCM, (2) DRAM capacity, and (3) performance.

6.7.1 PCM Writes

Figure 6.9 reports the number of PCM writes normalized to KG-N. This baseline reduces writes compared to a PCM-only main memory system by 75% (not shown), which results in improved, but still impractical PCM write rates (which are shown in Figure 6.16). KG-W reduces the number of PCM writes compared to KG-N by 45% on average. The GraphChi applications, on average, write to PCM more often than other application groups, even with KG-W. This is because they allocate more large objects directly in PCM than other application groups.

CGZ-S-F1 is the most effective configuration in reducing PCM writes. It eliminates 65% and 30% more PCM writes compared to KG-N and KG-W, respectively. CGZ-D1 reduces the number of PCM writes compared to KG-N for the DaCapo benchmarks (by 23%) and Pjbb (by 31%), but not the GraphChi workloads because CGZ-D1 prioritizes small objects, while putting more large objects in PCM. CGZ-F1 does slightly worse than KG-W on average, reducing the number of PCM writes by 35%. CGZ-F1 leads to an increase in PCM writes over KG-W for many DaCapo applications as a result of the lack of a survivor space in front of the mature space as in KG-W. However, CGZ-F1 is more effective than KG-W at eliminating PCM writes for the GraphChi applications because it leverages ahead-of-time information about the write behavior of large objects. Adding a survivor space greatly reduces the number of PCM writes. CGZ-S-D1 reduces the number of PCM writes by 54%. Finally, CGZ-S-F1 eliminates the largest number of PCM writes. Although both CGZ-S-F1 and CGZ-S-D1 reduces PCM writes significantly compared to KG-N, CGZ-S-D1 saves more DRAM capacity than CGZ-S-F1 at the expense of PCM writes.

We conclude that Crystal Gazer eliminates a large number of PCM writes compared to KG-N, on par with or significantly surpassing KG-W. These large reductions are a result of ahead-of-time profiling of application for write-intensive allocation sites.

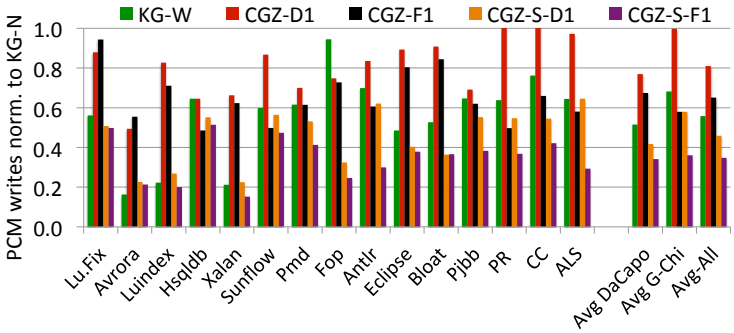


Figure 6.9: Number of PCM writes normalized to KG-N. *CGZ reduces the number of PCM writes, especially with a survivor space.*

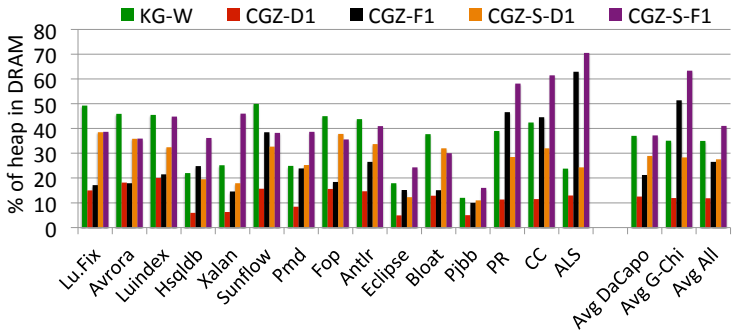
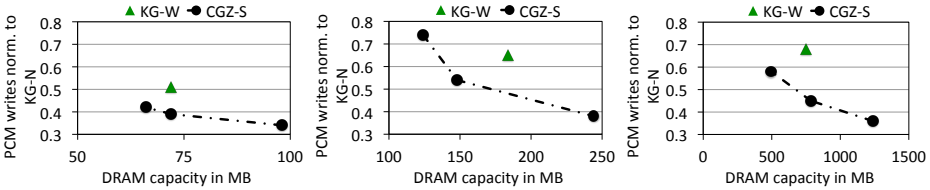


Figure 6.10: Fraction of heap allocated in DRAM. *CGZ reduces DRAM capacity needs compared to KG-W, especially without a survivor space.*

6.7.2 DRAM Capacity

DRAM will likely be a scarce resource in hybrid DRAM-PCM systems, which motivates minimizing the use of DRAM. Because KG-N stores only newly allocated nursery objects in DRAM, it consumes the least amount of DRAM among the collectors: only 4 MB for DaCapo and Pjbb, and 32 MB for GraphChi. All the other collectors trade reduced writes to PCM for DRAM capacity. Figure 6.10 shows the percentage of the heap that the CGZ and KG-W collectors allocate in DRAM. We take a snapshot of the heap at every collection cycle and compute the average heap volume (in MB) in DRAM and PCM. KG-W and CGZ-S-F1 allocate the largest fraction of the heap in DRAM: 35% and 41%, respectively. CGZ-S-F1 pays this price to reduce the number of PCM writes the most, as discussed in the previous section. CGZ-S-D1 reduces the use of DRAM, but incurs the space cost of the DRAM survivor space, placing 28% of the heap in DRAM. Finally, CGZ-D1 consumes the least amount of DRAM, 12% on average (a reduction of 23% compared to KG-W), but incurs more PCM writes. In terms of MB of DRAM consumed, CGZ-D1 only requires 40 MB compared to 132 MB for KG-W on average for our workloads, a reduction of 68%.



(a) DaCapo (b) Pjbb (c) GraphChi

Figure 6.11: Pareto-optimal configurations for CGZ-S compared to KG-W in terms of PCM writes versus DRAM capacity. *CGZ provides the flexibility of trading off PCM writes for DRAM capacity and vice versa.*

6.7.3 Trading Off PCM Writes and DRAM Capacity

Combining the results in Figure 6.9 and 6.10 illustrates how the different CGZ collector configurations trade off fewer PCM writes (Figure 6.9, left to right decreases) for increases in DRAM usage (Figure 6.10, left to right increases). The DENS heuristic reduces allocation in DRAM at the cost of an increased number of PCM writes. The FREQ heuristic on the other hand, increases DRAM allocation while reducing the number of PCM writes. Figure 6.11 visualizes this tradeoff by reporting normalized PCM writes (to KG-N) versus DRAM capacity for a number of Pareto-optimal configurations for CGZ-S by setting different thresholds for the different heuristics on a per-application basis. The key take-away point from these results is that Crystal Gazer offers a set of Pareto-optimal tradeoffs between PCM writes and DRAM capacity, and that KG-W is sub-optimal compared to CGZ-S, i.e., KG-W incurs more PCM writes for the same DRAM capacity and/or requires more DRAM capacity for the same number of PCM writes. Furthermore, KG-W offers no tradeoffs in DRAM capacity versus PCM writes, only offering a single operating point.

We can further configure CGZ collector thresholds to control the PCM write and DRAM capacity tradeoff. To minimize writes to PCM (the right-most points in Figure 6.11), we set θ_h to 1% and θ_f to 1 for FREQ. For minimum DRAM usage (the left-most points in Figure 6.11), we set θ_h to 1% for DaCapo and 25% for Pjbb and GraphChi, and use DENS with θ_d set to 1 for all applications. We observe that setting θ_f between 5K and 50K also results in Pareto-optimal configurations. In general, increasing θ_h and θ_f minimizes DRAM usage but increases PCM writes. The best θ_d ranges between 0.1 and 1.

6.7.4 Allocation Site Analysis

To better understand write-intensive objects, we classify them with the F1 heuristic into four categories based on the location of the allocation site: (1) gnu libraries, (2) Java class libraries, (3) Jikes RVM class files, and (4) application-specific code. Figure 6.12 shows that application code allocates 58% of write-intensive objects; the Java class libraries allocate 28% of them; Jikes class files 12%, and gnu libraries only 2%. We observe a similar breakdown for D1. Since application-specific code allocates most write-intensive objects, this further motivates profiling applications for

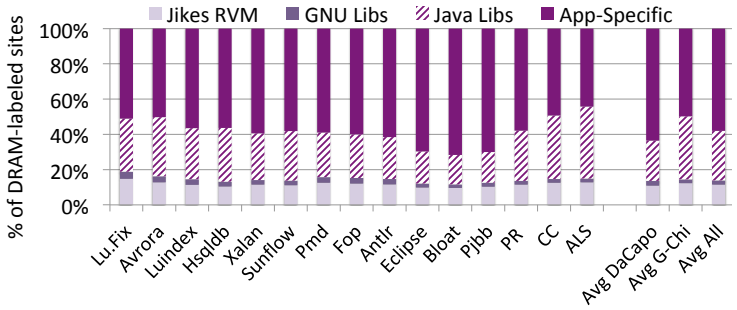


Figure 6.12: Breaking down the *DRAM*-labeled allocation sites into four categories: `gnu` libraries, Java libraries, Jikes RVM class files, and application code. *The application-specific class files allocate the majority of write-intensive objects.*

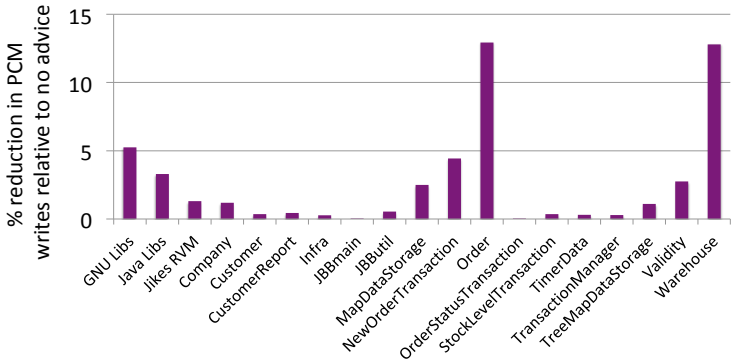


Figure 6.13: Understanding the reduction in PCM writes on a per allocation site basis for Pjbb. *Keeping the objects allocated from sites in the application-specific class files,* `Order` *and* `Warehouse`*, leads to the greatest reduction in PCM writes.*

write-intensive objects.

Next, we show the reduction in PCM writes by selectively labeling certain allocation sites as DRAM for Pjbb. The goal is to understand which allocation sites result in the largest reductions in PCM writes. Figure 6.13 breaks down the contributions of various allocation sites to PCM write reductions. We report the reduction in PCM writes with CGZ-S-F1 compared to a baseline that uses no advice, i.e., no allocation site is labeled as DRAM. We observe that allocation sites in the application-specific class files are the greatest source of PCM writes, and labeling them as DRAM saves the most writes to PCM. Specifically, `gnu` libraries, Java class libraries, and Jikes class files, together reduce writes to PCM by up to a maximum of 5%. On the other hand, the two application-specific class files, `Order` and `Warehouse`, each reduce 13% of writes to PCM relative to using no advice at all with CGZ-S-F1.

The results for Pjbb in this section, showing that application-specific class files are the greatest source of PCM writes, are also representative of other benchmarks we evaluate in this work.

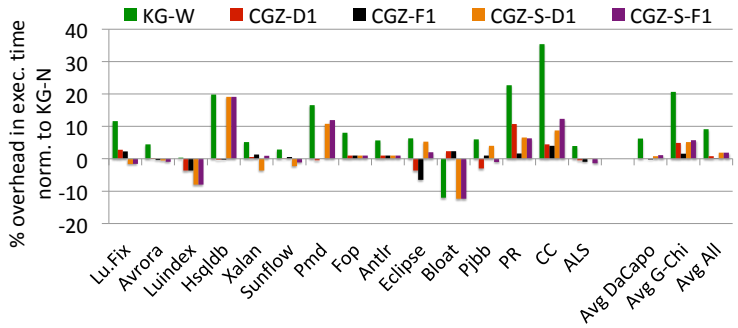


Figure 6.14: Execution time normalized to KG-N. *CGZ incurs negligible execution time overhead compared to KG-N.*

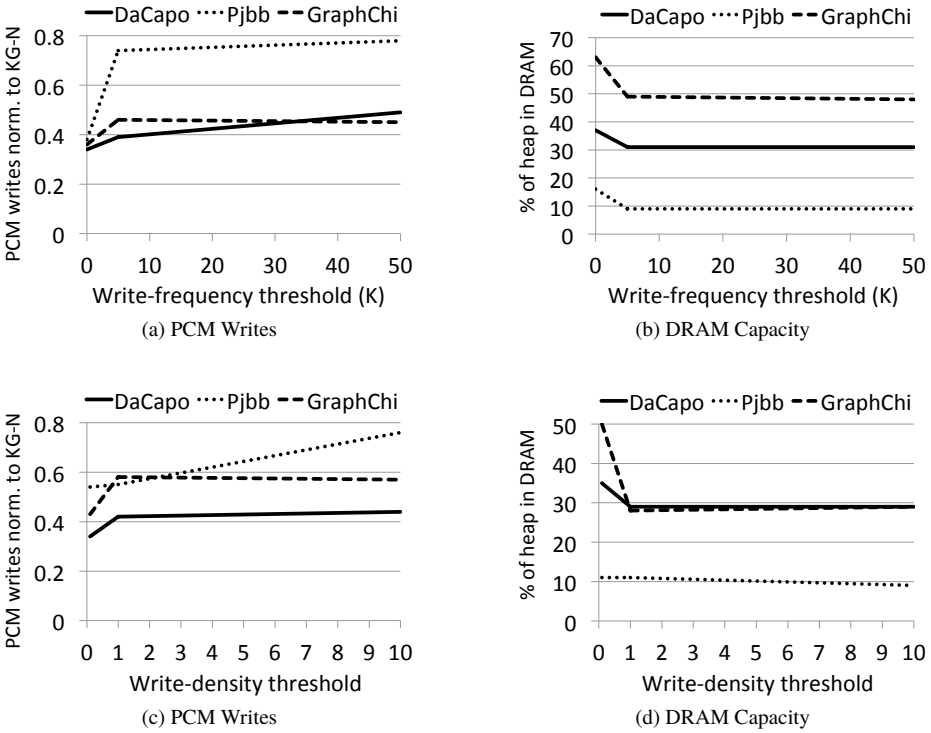
6.7.5 Performance

All hybrid memory systems will incur overhead due to the higher latencies to PCM versus DRAM, which our emulation infrastructure does not accurately capture, but was explored in simulation previously [2]. This section quantifies the performance overheads in Crystal Gazer compared to previously proposed Kingsguard write-rationing garbage collectors.

Figure 6.14 reports execution time normalized to KG-N. KG-W incurs an average execution time overhead of 9% over KG-N (and up to 35%). The main reason for this overhead is the extra code KG-W executes in the write barrier to monitor object writes, whereas KG-N has no monitoring. KG-N simply promotes all objects to PCM.

CGZ eliminates all of the overhead of KG-W for most applications. The best CGZ configurations (CGZ-F1 and CGZ-D1) improve performance over KG-W by 8% on average. hsqldb is notably better than KG-W with 20% reduction in execution time for CGZ-F1. CGZ also reduces the execution time of eclipse compared to KG-N by 3%. However, some applications do incur a slight performance degradation as shown in Figure 6.14. The reasons include: (1) setting the DRAM_BIT for objects predicted as highly written, and (2) checking whether the DRAM_BIT is set during nursery collection. This degradation is particularly prominent for two of the GraphChi applications: PR and CC. We observe only slight performance differences between the F1 and D1 configurations which are the result of a different number of objects being placed in DRAM versus PCM.

CGZ-S also eliminates the monitoring overheads in KG-W, but has higher overhead for some applications (e.g., hsqldb) than CGZ because of the additional survivor space collections. On the other hand, CGZ-S places more objects in DRAM than CGZ. The overhead is limited to less than 2% on average compared to CGZ, putting CGZ-S on par with KG-N. One benchmark, namely bloat, performs better with a survivor space compared to CGZ. Survivors space collections when running bloat preclude full-heap collections by giving objects more time to die in the survivor space. This reduces the total garbage collection time in bloat which translates to an overall performance improvement. Overall, profile-driven garbage collection brings down the high execution time overhead of KG-W to a level similar to KG-N.



(a) PCM Writes (b) DRAM Capacity

(c) PCM Writes (d) DRAM Capacity

Figure 6.15: Showing the impact on PCM writes and DRAM capacity from changing the write-frequency threshold (a and b) and write-density threshold (c and d). *Increasing the write-frequency and write-density thresholds increases PCM writes but reduces DRAM space usage.*

6.7.6 Sensitivity Analyses

Figures 6.15 (a) and (b) show PCM writes and DRAM capacity as we vary θ_f from 1 to 50K. PCM writes normalized to KG-N increase as we increase θ_f. We observe an increase in PCM writes up to a θ_f of 5K for all three application groups. From 5K to 50K, PCM writes stabilize. Conversely, the percentage of heap in DRAM shows a decrease up to 5K. Beyond that, increasing θ_f does not impact DRAM capacity much.

Figures 6.15 (c) and (d) show PCM writes and DRAM capacity as we vary θ_d from 0.1 to 10. Increasing θ_d beyond 1 has limited impact on PCM writes normalized to KG-N on average for DaCapo and GraphChi applications. Individual applications from the two suites sometimes show different trends. Conversely for Pjbb, PCM writes increase linearly as we increase θ_d from 0.1 to 10. The impact on the percentage of heap in DRAM is less prominent for Pjbb. This is because a small percentage of objects are responsible for most writes to the mature heap.

Table 6.2: Total allocation, the heap sizes of our applications, the survival rates of nursery and survivor spaces in CGZ-S, and the average and maximum DRAM usage in MB for KG-W and six CGZ-S configurations. The homogeneity threshold is fixed to 1%. Three write-frequency thresholds of 1, 5 K, and 50 K, and three density-thresholds of 0.1, 1, and 10, are used to show the DRAM space occupancy of CGZ-S.

	allocation MB (1)	Heap MB (2)	CGZ-S nursery survival % (3)	CGZ-S survivor survival % (4)	DRAM KG-W avg (5)	DRAM KG-W max (6)	DRAM θ_f = 1 θ_h = 1% avg (7)	DRAM θ_f = 1 θ_h = 1% max (8)	DRAM θ_f = 5K θ_h = 1% avg (9)	DRAM θ_f = 5K θ_h = 1% max (10)	DRAM θ_f = 50K θ_h = 1% avg (11)	DRAM θ_f = 50K θ_h = 1% max (12)	DRAM θ_d = 0.1 θ_h = 1% avg (13)	DRAM θ_d = 0.1 θ_h = 1% max (14)	DRAM θ_d = 1 θ_h = 1% avg (15)	DRAM θ_d = 1 θ_h = 1% max (16)	DRAM θ_d = 10 θ_h = 1% avg (17)	DRAM θ_d = 10 θ_h = 1% max (18)
Lusearch	4294	68	4%	29%	8	11	8	10	7	10	7	10	7	10	7	10	8	10
Lu.Fix	848	68	2%	25%	11	15	9	14	9	14	9	14	9	14	9	14	9	14
Avrora	64	98	15%	0%	12	16	10	14	10	14	10	14	10	14	10	14	10	14
Luindex	37	44	22%	0%	12	14	12	14	11	13	11	13	12	14	9	11	9	12
Hsqldb	165	254	60%	88%	16	22	27	37	15	21	15	21	27	37	14	21	14	20
Xalan	980	108	14%	9%	14	19	29	58	16	27	11	17	20	37	11	17	11	17
Sunflow	1920	108	2%	13%	12	16	10	17	9	14	9	15	10	17	9	14	9	14
Pmd	364	98	23%	68%	19	27	22	32	15	24	16	24	16	23	15	22	16	25
Pmd.S	202	98	27%	47%	14	19	16	25	13	20	13	21	13	21	12	19	13	21
Fop	56	80	20%	82%	12	16	11	15	12	15	12	16	11	15	12	15	11	15
Antlr	246	48	15%	0.16%	9	14	11	16	11	16	12	17	12	17	9	14	9	14
Eclipse	3082	160	14%	37%	19	25	29	51	18	26	17	24	22	34	15	22	15	21
Bloat	1246	66	4%	19%	12	16	11	16	11	16	11	16	11	16	12	17	11	16
Avg DaCapo	1040	100	17%	32%	13	18	16	25	12	18	12	79	14	21	11	16	11	16
Heap %					30%		35%		30%		30%		33%		28%		28%	
Pjbb	2314	400	20%	84%	32	46	40	61	23	31	23	52	28	37	28	37	21	31
Heap %					12%		16%		9%		9%		11%		11%		9%	
PR	6946	512	36%	99%	97	225	140	321	88	174	90	177	96	202	73	136	72	138
CC	5507	512	24%	97%	99	225	134	347	93	180	92	169	91	185	72	135	72	129
ALS	14245	512	10%	63%	66	113	191	260	182	241	180	242	187	249	65	108	65	110
Avg GraphChi	9000	512	23%	86%	87	188	155	309	121	198	121	196	125	212	70	126	70	126
Heap %					35%		63%		49%		48%		50%		28%		29%	
Avg All	2900	206	17%	46%	27	49	42	77	32	50	32	37	34	55	22	37	22	37
Heap %					30%		46%		34%		34%		37%		24%		24%	

6.7.7 Memory and Demographic Analysis

Table 6.2 reports object demographics and shows the average and maximum DRAM usage in MB for KG-W and different CGZ-S configurations for all of the benchmarks considered in this study. The total allocation of our applications varies, between 56 MB and 14 GB of memory (column 1). The GraphChi applications in particular allocate more than DaCapo and Pjbb. The average nursery survival rate of our applications is 17% and a maximum of 66% (column 3). We show the survival rate of objects in the survivor space in CGZ-S in column 4. A number of applications, such as xalan and bloat, benefit greatly from a survivor space, as it gives many objects a chance to die in DRAM. For instance, only 8% of objects in xalan are promoted to the mature space; which means an even smaller percentage of objects are written to PCM.

The remaining columns show the average and maximum DRAM space occupancy in MB for KG-W (column 5 and 6) and six CGZ-S configurations (column 7 through 18). Specifically, we show the DRAM space occupancy for different θ_f and θ_d. We also show the percentage of heap in DRAM across the three benchmark suites and overall for all applications. To calculate the percentage heap in DRAM, we first measure the average DRAM and PCM space occupancy in MB. The sum of the two spaces (MB) is the total heap used by the application. The percentage of total heap in DRAM is shown in Table 6.2 as Heap %.

We observe that CGZ-S-D10 maximizes the use of PCM for many applications. Conversely, CGZ-S-F1 maximizes the use of DRAM to eliminate the largest number of writes to PCM. In particular for GraphChi applications, CGZ-S-F1 places more than 200 MB per application instance in DRAM. Thus, by prioritizing small objects, the density heuristic minimizes DRAM usage for modern graph analytic workloads, with large heaps and high allocation rates.

6.7.8 PCM Lifetime and Write Rates

This section analyzes the PCM write rate results and their implications for PCM lifetime. As Chapter 5 showed, eliminating PCM writes improves PCM lifetime. PCM lifetime in years depends on its write rate and cell endurance. Prototype PCM has a cell endurance between 10 M and 100 M writes per cell [81, 7]. All of our results assume hardware wear-leveling is enabled. As writes to PCM reduce, so does the write rate, but the write rate is also inversely proportional to execution time. Optimizing only for write rate would thus lead to incorrect conclusions. As an example, turning off compiler optimizations to make the program execute slower would decrease write rate. Because execution time depends on a number of factors including on-chip cache sizes, number of threads, cores, and garbage collection algorithm, it is not meaningful to directly compare normalized write rates.

Figure 6.16 shows the absolute PCM write rates in MB/s that we observe on our emulation platform. Since PCM hardware is still evolving, the write rates on future PCM hardware may differ from our measurements. Recent work uses a real PCM prototype to evaluate hybrid memory at Facebook [42]. Their work shows that hardware vendors limit the number of times the entire PCM memory (or drive) can

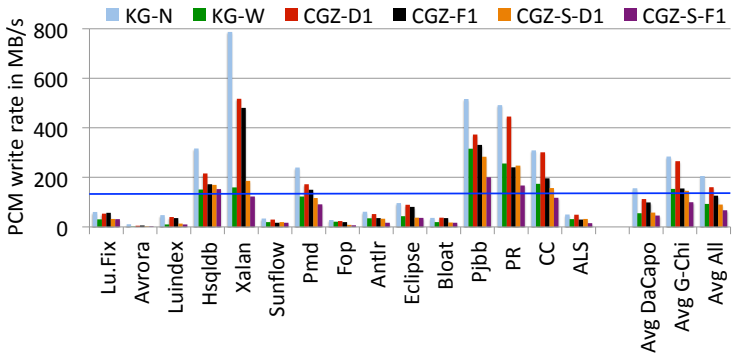


Figure 6.16: PCM write rates in MB/s for all of our benchmarks using various write-rationing garbage collectors. *Profile-driven write-rationing garbage collection makes PCM a practical DRAM replacement by significantly reducing its write rates.*

be written per day. The specific metric is called drive writes per day (DWPD). The most recently reported DWPD for a 375 GB NVM drive is 30 [42, 52]. This DWPD results in a recommended write rate of 140 MB/s (blue horizontal line in Figure 6.16). We observe in Figure 6.16 that all write-rationing collectors significantly reduce the write rates and many are brought below the recommended rate to make PCM practical as main memory. In particular, CGZ-S-F1 limits the PCM write rates of all but 3 workloads to below the recommended rate, while improving the performance over KG-W at the same time.

We observe that many workloads write at a rate that is still not practical for PCM. For instance, two of the graph applications have write rates above 140 MB/s, even with CGZ-S-F1. Furthermore, future servers with more cores will likely run many more applications in parallel, which will result in even higher write rates. This necessitates more research in software approaches to bring write rates down even further. Some reduction will come from innovations at the device and architecture level, or from using hybrid memories where some writes could be guided to DRAM. Due to the nature of PCM material, software has a greater role to play in making PCM practical as (part of) main memory.

6.8 Summary and Interpretation

This chapter demonstrates that profile-driven write-rationing garbage collection can improve PCM's lifetime in hybrid memories while limiting consumption of DRAM capacity. Crystal Gazer overcomes the shortcomings of the prior state-of-the-art write-rationing garbage collector, KG-W, which dynamically monitors writes to nursery survivors to decide whether to promote to mature DRAM or mature PCM. Crystal Gazer improves the accuracy and reduces the cost of write-rationing garbage collection by predicting object write-intensity based on offline allocation site profiling. It copies nursery survivors to mature DRAM if predicted highly written or to mature PCM if predicted read-mostly. Using a survivor space in-between the nursery and mature space further reduces the number of writes to PCM at the cost of increasing DRAM capacity

consumption. Because allocation site prediction of write-intensity is highly accurate, our static technique out-performs the dynamic techniques used by the Kingsguard KG-W collector. We demonstrate that Crystal Gazer, by changing the heuristics and thresholds, provides Pareto-optimal operating points in terms of PCM lifetime and DRAM capacity. Our experimental results use emulation on real hardware and show that Crystal Gazer significantly improves performance compared to the state-of-the-art, KG-W, while reducing the number of PCM writes when optimized for PCM lifetime, and requiring less DRAM capacity when optimized for the smallest DRAM capacity.

This work targets PCM write endurance. Another disadvantage of the PCM technology is its high access latency. In this work, we are concerned with the reduction in PCM writes and overheads of Crystal Gazer, and also improving upon Kingsguard. Therefore, PCM access latency does not affect the conclusions of this work. Regardless, we made an effort to accurately model PCM access latency on our emulation platform. Our solution to introduce interference in the remote socket to slow down remote (PCM) memory accesses lead to non-determinism in the execution time results, and thus we removed it for the final experiments. Instead of using a simulator to model PCM latency, we believe emulation is a more valuable way to do the evaluation of hybrid memories. Emulation includes real system effects (advanced caching, prefetching, memory bandwidth resource contention, etc.) that no simulator can accurately model and lets us explore many more software configurations and bigger workloads in the same resource budget.

In our evaluation, we use different input datasets for training and production experiments. The benchmarks from the DaCapo suite have default and large input datasets. For Pjbb and GraphChi, we experiment with a range of newly created datasets for production runs. We show results with one dataset. Results always depend on the specific dataset and application, and the profiling correctly predicting the allocation sites' write-intensity even if the input changes.

Crystal Gazer could be extended to better tolerate PCM's high access latencies. One future work could be to not only guide highly written objects away from PCM, but also to find highly accessed objects and place them in DRAM to reduce their access latency. In this way, we could use garbage collection to fight both of the drawbacks of PCM, resulting in a high-performance hybrid memory system with a long lifetime.

Our profiling in this work is architecture independent, i.e., in the profiling step we do not track whether the write operation hits or misses in the cache. Therefore, our classification of allocation sites as DRAM is conservative. A DRAM-labeled site that allocates objects which have high cache locality needlessly wastes DRAM space as writes to these objects hit in the cache. Unfortunately, current hardware precludes correlating allocation sites to cache misses, making it difficult to incorporate cache effects in the profiling step. Nevertheless, our PCM writes and execution time results with Crystal Gazer increase our confidence in the accuracy of our current profiling approach.

Our Crystal Gazer collectors by default promote objects that outlive a nursery or a survivor collection to PCM. If advice is not available, or highly written objects are promoted to PCM due to misclassification of allocation sites, our Crystal Gazer collectors lack a dynamic approach to move objects away from PCM. When advice

is not available, KG-W is the best approach. Future work should also consider dynamic call site write-prediction, similar to dynamic call site lifetime-prediction [68]. Production systems will require some dynamic monitoring of PCM objects or memory, perhaps by the OS, to recover from changes in behavior, bad predictions, and malicious write attacks.

Profile-driven write-rationing garbage collection makes PCM a practical DRAM replacement by aggressively reducing writes to it in a hybrid memory setting. It requires minimal OS support and enables the use of PCM for commodity applications without changes to the programming language or model.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Contemporary semiconductor scaling trends are the prime force behind emerging hardware heterogeneity. Dennard scaling that enabled greater transistor counts at constant power density has stopped. Energy efficiency is now a first-order concern in processor design and operation. On the main memory side, manufacturing complexity has encouraged hybrid memories that combine DRAM and PCM to deliver high performance and capacity. Software must take advantage of emerging hardware heterogeneity to optimize critical metrics such as performance and energy efficiency.

On the software side, programmers prefer managed languages because of their productivity advantages. Managed runtimes facilitate the execution of managed applications by providing services such as garbage collection. Managed runtime environments contain rich semantic information about application behaviors. This thesis asks the question, question,"how can we exploit the semantic information in a managed language runtime to improve the utility of heterogeneous processors and memories."

We believe managed runtimes can help exploit heterogeneous processors and emerging hybrid memory better than existing hardware and OS approaches. This thesis discusses two ways in which using semantic information in the managed runtime improves the performance and energy efficiency of heterogeneous multicore processors. Furthermore, this thesis presents two write-rationing garbage collectors that better manage hybrid memories consisting of DRAM and PCM than prior approaches.

This thesis presents GC-criticality-aware scheduling that uses semantic information from the managed runtime to optimally schedule application and concurrent GC threads on heterogeneous multicores. We first show that GC-criticality exists in popular Java applications. Specifically, on a heterogeneous multicore processor, GC can become critical if always left to run on the lower power cores with the intent to save energy. When GC becomes critical, it stops the application to free unused memory. This stop-the-world pause hurts both performance and energy efficiency. We further show

that, contrary to intuition, GC benefits from running on the big (out-of-order) cores of a heterogeneous multicore. Thus, when GC becomes critical, it is vital to execute it on the big cores. Our proposed GC-criticality-aware scheduling informs the OS scheduler to boost the priority of GC threads for the big cores by sending a criticality signal. The OS alone is not able to predict GC-criticality. Our algorithm is performance and energy-neutral for GC-uncritical Java applications, and significantly speeds up GC-critical applications: by 16% on average, while being 20% more energy-efficient for a heterogeneous multicore with three big cores and one small core.

Multicore processors with homogeneous cores similar in their architectural capabilities have a different form of heterogeneity. Each core has DVFS that enables changing a core's voltage and frequency to save energy. An accurate performance predictor guides a DVFS energy manager to choose the best DVFS setting based on a user-specified slowdown threshold. Prior DVFS predictors are only accurate for single-threaded native application written in C and C++. Multithreaded managed applications such as Java introduce two additional challenges for performance prediction. First, synchronization in multithreaded applications leads to inter-thread dependences. Thus, changing the frequency of one core (thread) impacts the execution of dependent threads. Managed applications also issue a burst of store operations due to memory management. Prior predictors ignore store operations assuming they are not on the critical path. This thesis proposes *DEP+BURST*, an accurate DVFS performance predictor for managed multithreaded applications. DEP+BURST intercepts the synchronization activity in applications and then uses analytical modeling to reconstruct the execution at different frequency settings as the application executes at a specific frequency. It uses a new hardware performance counter to track the time the core stalls due to waiting for store operations to resolve. This counter allows DEP+BURST to model the performance impact of store bursts accurately. Our predictor lowers the performance estimation error from 27% for a state-of-the-art predictor to 6% on average, for a set of multithreaded Java applications when the frequency is scaled from 1 to 4 GHz. We then develop a new energy manager to optimize total system energy, achieving an average reduction of 15.6% for a set of Java benchmarks.

Non-volatile memory (NVM) technologies offer more scalability than DRAM. The most promising NVM is phase-change memory (PCM). PCM is byte-addressable and offers abundant main memory capacity. Its main disadvantages are high access latency and limited write endurance. Hybrid memory combines DRAM and NVM to provide the best of both DRAM and PCM. Hardware wear-leveling mitigates PCM wear-out by spreading writes out across the entire PCM capacity. Prior OS solutions move *coarse-grained* pages in DRAM to limit PCM writes and improve its lifetime. Unfortunately, prior approaches are reactive and their DRAM usage is excessive. This thesis shows that popular Java applications can wear PCM out in 4 years or less.

This thesis proposes two write-rationing garbage collectors for hybrid memories that aim to mitigate PCM wear-out and improve its lifetime. Write-rationing garbage collectors keep frequently written objects in DRAM to mitigate PCM wear-out. They keep read-mostly objects in PCM to exploit its capacity. This thesis first proposes two Kingsguard collectors. They exploit the observation that most writes in Java applications occur to young (nursery) objects and a small fraction of mature objects. Kingsguard-nursery (KG-N) places the nursery in DRAM. It copies the surviving

nursery objects in PCM. KG-N increases PCM lifetime by 5× over PCM-Only. Kingsguard-writers (KG-W) copies nursery survivors to an observer space in DRAM. It monitors all the mature objects for writes and keeps frequently written objects in DRAM. KG-W reduces PCM writes over KG-N by 2× which translates to a 2× improvement in PCM lifetime in hybrid memories. Dynamic monitoring incurs a 9% performance overhead on average for 15 Java applications.

This thesis finally proposes profile-driven write-rationing garbage collection to eliminate the performance overhead of dynamic monitoring in KG-W. More specifically, this thesis proposes Crystal Gazer (CGZ) that uses offline profiling to identify frequently written objects in Java applications. Our research shows that writes in Java applications are predictable on a per allocation-site basis. We profile individual object writes and their allocation sites. We use heuristics to classify sites as highly written (DRAM) or read-mostly (PCM). Crystal Gazer uses the profile during execution time to guide object placement in DRAM and PCM. It first allocates objects in a DRAM nursery. It labels objects at allocation time as coming from a highly written or read-mostly site. During a nursery collection, it copies objects to DRAM or PCM based upon the predicted write-intensity label. Leveraging profile information eliminates the performance overhead of dynamic monitoring. CGZ also eliminates 30% more PCM writes compared to KG-W when optimized for extending PCM lifetime.

This thesis shows using semantic information in the managed runtime can exploit heterogeneous hardware better. The hardware and the OS in isolation can not always form the best policies to exploit heterogeneity. This thesis shows how to determine and communicate the criticality of garbage collection in managed runtimes happening concurrently with the application. Accurate performance predictors for multithreaded applications requires communicating synchronization activity from the runtime environment to the DVFS management framework. Finally, write-rationing garbage collection opens up a promising avenue for managing hybrid DRAM-PCM memories more efficiently than prior hardware and OS approaches.

7.2 Future Work

This section discusses various avenues for future work, both on the side of exploiting heterogeneous processors for managed applications, and exploiting garbage collection to manage hybrid DRAM-PCM memories.

7.2.1 Scheduling for Heterogeneous Multicores

This thesis explores how best to schedule managed applications that use concurrent GC on heterogeneous multicores. We show the benefits of GC-criticality-aware scheduling. Multiprogrammed workloads consisting of multiple applications are increasingly common, especially with the popularity of cloud computing. An avenue for future work is scheduling multiprogrammed managed applications on heterogeneous multicores. GC-criticality-aware scheduling for multiprogrammed workloads should take into account the priority of individual applications.

In this thesis, we consider Java applications from the DaCapo benchmark suite. Several recent big-data platforms are also written in a managed language. Examples include Apache Spark and Apache Hadoop. These platforms use large heaps and stress the garbage collector more than DaCapo applications. Future work should consider scheduling these big-data applications on heterogeneous multicores. Server processors today use homogeneous multicores; however, each core still has DVFS. Future work should consider DVFS to boost the priority of GC threads.

Today's software stacks are increasingly multi-layer in nature. Consider a managed application that executes on top of a virtualized host (e.g., VMware Workstation) and the host runs a Linux OS. Future work should consider how best to schedule such versatile runtime environments on top of heterogeneous multicores.

7.2.2 Performance Prediction

This thesis explores DVFS performance prediction for multithreaded managed applications with the aim to save energy. We use a stop-the-world GC in our evaluations. Predicting the performance impact with concurrent GC is more challenging and an avenue for future work. Concurrent GC runs asynchronously with the application. Changing the frequency of concurrent GC may (or may not) make it critical. When GC becomes critical, it slows down the application. Future work should investigate predicting the criticality of concurrent GC due to changes in DVFS. More generally, asynchronous patterns in modern software are increasingly common and make performance prediction more challenging. Future work should research performance prediction for a broad set of programming patterns and environments.

This thesis focuses on accurate DVFS performance prediction. Predicting the performance impact of changing the core type on a heterogeneous multicore for multithreaded managed applications is even more challenging. An interesting avenue for future work is fast and accurate analytical models for online prediction of changing core types on a heterogeneous multicore.

7.2.3 Write-Rationing Garbage Collection

This thesis proposes two write-rationing garbage collectors that aim to mitigate PCM wear-out and improve its lifetime. The collector configurations that optimize PCM lifetime perform two copies of long-lived objects, from the nursery to a DRAM space for nursery survivors, and from that survivor space to either DRAM or PCM. This extra copying has two disadvantages: (1) it hurts performance, and (2) it necessitates updating PCM object references to copied objects leading to PCM writes. Prior work proposes allocation site based prediction of object lifetimes for Java workloads [19, 15]. Object lifetime prediction used with pretenuring can eliminate the extra copying of long-lived objects. The mutator can directly allocate objects from allocation sites that produce long-lived objects in the mature DRAM or PCM. The resulting system is likely to use the DRAM and PCM spaces more efficiently. Another promising avenue to eliminate the extra copying in CGZ-S is to explore non-moving variants of Immix [18]. These non-moving Immix collectors trade the locality of contiguous

allocation for increased heap occupancy and fragmentation. This tradeoff is likely a good one for PCM lifetimes and merits further exploration but may increase DRAM requirements.

The default Immix collector defragments blocks based on a threshold that indicates fragmentation is preventing the collector from using some fraction of the memory in partially filled blocks in the mature space. Immix defragmentation combines marking with copying based on occupancy statistics of the partially filled blocks, seeking to move the fewest numbers of objects and to create the maximum number of completely free blocks. It thus trades increased numbers of writes to reduce memory consumption. PCM prefers exactly the opposite tradeoff – PCM is write-limited coupled with plentiful capacity. For the heap sizes this thesis explores, Immix defragmentation was never triggered. However in the limit, the collector should monitor and limit extreme fragmentation. This exploration of non-moving collectors and defragmentation is interesting avenue for future work.

The Kingsguard collector that reduces PCM writes the most, namely Kingsguard-writers, dynamically monitors mature object writes. Dynamically monitoring writes incurs a performance penalty. The Crystal Gazer collectors use offline profiling of allocation sites to deliver good performance and eliminate PCM writes, similar or more than Kingsguard-writers. Crystal Gazer has the drawback that for unprofiled applications, it defaults to copying nursery survivors to PCM. Future work should dynamically profile allocation sites to get the best of both approaches. More specifically, this new approach divides the execution of the application into a sampling phase and normal execution. The collector monitors mature objects writes only during the sampling phase. The compiler stores identifiers for allocation sites in the object headers. Together, this helps the collector to classify allocation sites as highly written or read-mostly during the sampling phase of execution. During the normal execution, the collector copies highly written objects to DRAM and the rest to PCM.

Emerging graph applications allocate nursery objects at high rates. Our proposed write-rationing garbage collectors isolate nursery writes in DRAM in a hybrid memory system. Future work should investigate approaches to mitigate the impact of high allocation rates on PCM's lifetime assuming a PCM-only system. Such approaches are necessary to make PCM a true DRAM replacement. On the application side, we should investigate techniques to reduce nursery allocation rates. The OS can map virtual memory with the aim to wear-level nursery writes across the entire PCM capacity. Other avenues of future work for PCM-only systems include cache partitioning to mitigate wear-out and approaches that improve cache locality to reduce PCM writes.

Bibliography

[1] Shoaib Akram, Jennifer B. Sartor, and Lieven Eeckhout. DEP+BURST: Online DVFS performance prediction for energy-efficient managed language execution. *IEEE Trans. Comput.*, 66(4), April 2017.

[2] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Write-rationing garbage collection for hybrid memories. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[3] Shoaib Akram, Jennifer B. Sartor, Kenzo Van Craeynest, Wim Heirman, and Lieven Eeckhout. Boosting the priority of garbage: Scheduling collection on heterogeneous multicore processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2016.

[4] Bowen Alpern, C. Richard Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Mark F. Mergen, Ton Ngo, James R. Russell, Vivek Sarkar, Mauricio J. Serrano, Janice C. Shepherd, Stephen E. Smith, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[5] Bowen Alpern, Steve Augart, Stephen M. Blackburn, Maria A. Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen J. Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark Mergen, J. Eliot B. Moss, Ton Anh Ngo, Vivek Sarkar, and Martin Trapp. The Jikes RVM Project: Building an open source research community. *IBM System Journal*, 44(2), 2005.

[6] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper.*, 19(2), 1989.

[7] Aravinthan Athmanathan, Milos Stanisavljevic, Nikolaos Papandreou, Haralampos Pozidis, and Evangelos Eleftheriou. Multilevel-cell phase-change memory: A viable technology. *IEEE J. Emerg. Sel. Topics Circuits Syst.*, 6(1), 2016.

[8] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1993.

[9] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the Conference on Computing Frontiers (CF)*, 2006.

[10] Abhishek Bhattacharjee and Margaret Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009.

[11] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffman, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Communications of the ACM*, 51(8):83–89, 2008.

[12] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2004.

[13] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2004.

[14] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2006.

[15] Stephen M. Blackburn, Matthew Hertz, Kathryn S. McKinley, J. Eliot B. Moss, and Ting Yang. Profile-based pretenuring. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(1), 2007.

[16] Stephen M Blackburn, Martin Hirzel, Robin Garner, and Darko Stefanović. pjbb2005: The pseudojbb benchmark. 2010.

[17] Stephen M. Blackburn and Antony L. Hosking. Barriers: Friend or foe? In *Proceedings of the International Symposium on Memory Management (ISMM)*, 2004.

[18] Stephen M. Blackburn and Kathryn S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[19] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinley, and J. Eliot B. Moss. Pretenuring for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[20] Santiago Bock, Bruce R. Childers, Rami Melhem, and Daniel Mossé. Concurrent page migration for mobile systems with os-managed hybrid memory. In *Proceedings of the ACM Conference on Computing Frontiers (CF)*, 2014.

[21] Geoffrey W. Burr, Matthew J. Breitwisch, Michele Franceschini, Davide Garetto, Kailash Gopalakrishnan, Bryan Jackson, BÃŒlent Kurdi, Chung Lam, Luis A. Lastras, Alvaro Padilla, Bipin Rajendran, Simone Raoux, and Rohit S. Shenoy. Phase change memory technology. *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena*, 28(2), 2010.

[22] J.A. Butts and G.S. Sohi. A static power model for architects. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2000.

[23] Dries Buytaert, Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. GCH: Hints for triggering garbage collections. *Trans. HiPEAC*, 1:74–94, 2007.

[24] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2012.

[25] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[26] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3), 2014.

[27] Jian Chen and L.K. John. Efficient program scheduling for heterogeneous multi-core processors. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2009.

[28] Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. Profile-guided proactive garbage collection for locality optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[29] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1998.

[30] N. Chitlur, G. Srinivasa, S. Hahn, P.K. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, Li Zhao, N. Ijih, S. Subhaschandra, S. Grover, Xiaowei Jiang,

and R. Iyer. QuickIA: Exploring heterogeneous architectures on real prototypes. In *Proceedings of the High Performance Computer Architecture (HPCA)*, 2012.

[31] Kihwan Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times. In *Proceedings of the Design, Automation and Test in Europe (DATE)*, 2004.

[32] Cliff Click. Azul's experiences with hardware/software co-design. Keynote at ECOOP, 2009.

[33] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Prediction models for multi-dimensional power-performance optimization on many cores. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[34] Kenneth Czechowski, Victor W. Lee, and Jee Choi. Measuring the power/energy of modern hardware. In *Tutorial at the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[35] John Demme and Simha Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[36] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F. Wenisch, and Ricardo Bianchini. CoScale: Coordinating CPU and memory system DVFS in server systems. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[37] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[38] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management (ISMM)*, 2004.

[39] Kristof Du Bois, Stijn Eyerman, Jennifer B. Sartor, and Lieven Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013.

[40] Kristof Du Bois, Jennifer B. Sartor, Stijn Eyerman, and Lieven Eeckhout. Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013.

[41] T. Chad Effler, Adam P. Howard, Tong Zhou, Michael R. Jantz, Kshitij A. Doshi, and Prasad A. Kulkarni. On automated feedback-driven data placement in multi-tiered memory. In Mladen Berekovic, Rainer Buchty, Heiko Hamann,

Dirk Koch, and Thilo Pionteck, editors, *Architecture of Computing Systems (ARCS)*, 2018.

[42] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2018.

[43] Hadi Esmaeilzadeh, Ting Cao, Yang Xi, Stephen M. Blackburn, and Kathryn S. McKinley. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[44] Stijn Eyerman and Lieven Eeckhout. A counter architecture for online DVFS profitability estimation. *IEEE Transactions on Computers (TC)*, 59(11), November 2010.

[45] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A performance counter architecture for computing accurate cpi components. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[46] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying magic: High-level low-level programming. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2009.

[47] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Ottawa Linux Symposium*, 2002.

[48] Tiejun Gao, Karin Strauss, Stephen M. Blackburn, Kathryn S. McKinley, Doug Burger, and James Larus. Using managed runtime systems to tolerate holes in wearable memories. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[49] Soraya Ghiasi, Tom Keller, and Freeman Rawson. Scheduling for heterogeneous processors in server systems. In *Proceedings of the Conference on Computing Frontiers (CF)*, 2005.

[50] Peter Greenhalgh. Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7: Improving energy efficiency in high-performance mobile platforms. http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf, September 2011.

[51] J. Ha, M. Gustafsson, S.M. Blackburn, and K. S. McKinley. Microarchitectural characterization of production JVMs and Java workloads. In *IBM CAS Workshop*, 2008.

[52] Jim Handy. Examining 3D XPoint's 1,000 times endurance benefit. 2017.

[53] Timothy H. Heil and James E. Smith. Concurrent garbage collection using hardware-assisted profiling. In *Proceedings of the International Symposium on Memory Management (ISMM)*, 2000.

[54] Wim Heirman, Souradip Sarkar, Trevor E. Carlson, Ibrahim Hur, and Lieven Eeckhout. Power-aware multi-core simulation for early design stage hardware/-software co-optimization. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[55] Sebastian Herbert and Diana Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2007.

[56] Michael Hicks, Luke Hornof, Jonathan T. Moore, and Scott M. Nettles. A study of large object spaces. *SIGPLAN Not.*, 34(3), October 1998.

[57] Adrian Hoban. Designing realtime solutions on embedded intel architecture processors. 2010.

[58] Joel Hruska. Why ram prices are through the roof. 2018.

[59] Shiwen Hu and Lizy K. John. Impact of virtual execution environments on processor energy consumption and hardware adaptation. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, 2006.

[60] Jipeng Huang and Michael D. Bond. Efficient context sensitivity for dynamic analyses via calling context uptrees and customized memory management. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013.

[61] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: Improving mutator locality. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2004.

[62] ITRS. Internatial technology roadmap for semiconductors 2.0: Executive Report, 2015.

[63] Michael R. Jantz, Forrest J. Robinson, Prasad A. Kulkarni, and Kshitij A. Doshi. Cross-layer memory management for managed language applications. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[64] José A. Joao, Onur Mutlu, and Yale N. Patt. Flexible reference-counting-based hardware acceleration for garbage collection. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[65] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[66] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. Utility-based acceleration of multithreaded applications on asymmetric cmps. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013.

[67] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. 1996.

[68] Maria Jump, Stephen M. Blackburn, and Kathryn S. McKinley. Dynamic Object Sampling for Pretenuring. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 152–162, 2004.

[69] Patrick Kennedy. Why server asps are rising the 2017-2018 ddr4 dram shortage. 2018.

[70] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. Interval-based models for run-time DVFS orchestration in superscalar processors. In *Proceedings of the ACM International Conference on Computing Frontiers (CF)*, 2010.

[71] Wonyoung Kim, M.S. Gupta, Gu-Yeon Wei, and D. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[72] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2010.

[73] Chandra Krintz. Coupling on-line and off-line profile information to improve program performance. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2003.

[74] Chandra Krintz and Brad Calder. Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[75] Mark H. Kryder and Chang Soo Kim. After hard drives — what comes next? *IEEE Transactions on Magnetics*, 45(10), 2009.

[76] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003.

[77] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2004.

[78] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[79] Nagesh B. Lakshminarayana, Jaekyu Lee, and Hyesoon Kim. Age based scheduling for asymmetric multiprocessors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009.

[80] Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[81] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009.

[82] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-change technology and the future of main memory. *IEEE Micro*, 30(1), January 2010.

[83] Myoung-Jae Lee, Chang Bum Lee, Dongsoo Lee, Seung Ryul Lee, Man Chang, Ji Hyun Hur, Young-Bae Kim, Chang-Jung Kim, David H. Seo, Sunae Seo, U-In Chung, In-Kyeong Yoo, and Kinam Kim. A fast, high-endurance and scalable non-volatile memory device made from asymmetric ta2o5-x/tao2-x bilayer structures. *Nature Materials*, 10(3), 2011.

[84] Soyoon Lee, Hyokyung Bahn, and Sam H. Noh. Characterizing memory write references for efficient management of hybrid pcm and dram memory. In *Proceedings of the International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2011.

[85] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, 2014.

[86] Sheng Li, Jung Ho Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[87] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the ACM/IEEE Conference on Supercomputing (ICS)*, 2007.

[88] Tong Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-isa heterogeneous multi-core architectures. In *Proceedings of the High Performance Computer Architecture (HPCA)*, 2010.

[89] Kai Ma, Xue Li, Ming Chen, and Xiaorui Wang. Scalable power control for many-core architectures running multi-threaded applications. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2011.

[90] Martin Maas, Philip Reames, Jeffrey Morlan, Krste Asanović, Anthony D. Joseph, and John Kubiatowicz. Gpus as an opportunity for offloading garbage collection. In *International Symposium on Memory Management (ISMM)*, 2012.

[91] Micron. TN-41-01: Calculating memory system power for DDR3, 2007.

[92] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N. Patt. Predicting performance impact of DVFS for realistic memory systems. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[93] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[94] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[95] Numonym. Phase change memory, 2008.

[96] Nvidia. Variable SMP – a multi-core CPU architecture for low power and high performance. http://www.nvidia.com/content/PDF/tegra_white_papers/Variable-SMP-A-Multi-Core-CPU-Architecture-for-Low-Power-and-High-Performance.pdf, 2011.

[97] OpenJDK Group. Hotspot VM.

[98] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspot server compiler. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM)*, April 2001.

[99] Soyeon Park, Weihang Jiang, Yuanyuan Zhou, and Sarita Adve. Managing energy-performance tradeoffs for multithreaded applications on multiprocessor architectures. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2007.

[100] Moinuddin K. Qureshi. Pay-as-you-go: Low-overhead hard-error correction for phase change memories. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[101] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[102] Moinuddin K. Qureshi, Andre Seznec, Luis A. Lastras, and Michele M. Franceschini. Practical and secure pcm systems by online detection of malicious write streams. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[103] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009.

[104] R. Radhakrishnan, N. Vijaykrishnan, L.K. John, and A. Sivasubramaniam. Architectural issues in Java runtime systems. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2000.

[105] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2011.

[106] B. Rountree, D.K. Lowenthal, M. Schulz, and B.R. de Supinski. Practical performance prediction under dynamic voltage frequency scaling. In *Proceedings of the International Green Computing Conference and Workshops (IGCC)*, 2011.

[107] Jennifer B. Sartor and Lieven Eeckhout. Exploring multi-threaded java application performance on multicore hardware. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2012.

[108] Jennifer B. Sartor, Wim Heirman, Steve Blackburn, Lieven Eeckhout, and McKinley McKinley. Cooperative cache scrubbing. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[109] William J. Schmidt and Kelvin D. Nilsen. Performance of a hardware-assisted real-time garbage collector. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.

[110] Andre Seznec. A phase change memory as a secure main memory. *IEEE Computer Architecture Letters*, 9(1), Jan 2010.

[111] Rifat Shahriyar, Stephen M. Blackburn, Xi Yang, and Kathryn S. McKinley. Taking off the gloves with reference counting Immix. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2013.

[112] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. Hass: A scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev.*, 43(2), April 2009.

[113] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. Green governors: A framework for continuously adaptive DVFS. In *Proceedings of the International Green Computing Conference and Workshops (IGCC)*, 2011.

[114] Sadagopan Srinivasan, Li Zhao, Ramesh Illikkal, and Ravishankar Iyer. Efficient interaction between os and architecture in heterogeneous platforms. *SIGOPS Oper. Syst. Rev.*, 45(1):62–72, February 2011.

[115] Darko Stefanovic, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[116] Bo Su, Joseph L. Greathouse, Junli Gu, Michael Boyer, Li Shen, and Zhiying Wang. Implementing a leading loads performance predictor on commodity processors. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2014.

[117] Bo Su, Junli Gu, Li Shen, Wei Huang, J.L. Greathouse, and Zhiying Wang. PPEP: Online performance, power, and energy prediction framework and DVFS space exploration. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[118] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[119] TIOBE. TIOBE Index for January. https://www.tiobe.com/tiobe-index/, 2019.

[120] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE)*, 1984.

[121] David Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Trans. Program. Lang. Syst.*, 14(1), January 1992.

[122] Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In *Proceedings of the international conference on Parallel architectures and compilation techniques (PACT)*, 2013.

[123] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2012.

[124] Chenxi Wang, Ting Coa, John Zigman, Fang Lv, Yunquan Zhang, and Xiaobing Feng. Efficient management for hybrid memory in managed language runtime. In *Proceedings of the IFIP International Conference on Network and Parallel Computing (NPC)*, 2016.

[125] Wei Wei, Dejun Jiang, Sally A. McKee, Jin Xiong, and Mingyu Chen. Exploiting program semantics to place data in hybrid memory. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[126] Paul R. Wilson. A simple bucket-brigade advancement mechanism for generation-bases garbage collection. *SIGPLAN Not.*, 24(5), May 1989.

[127] Q. Wu, V.J. Reddi, Y. Wu, J. Lee, D. Connors, D. Brooks, M. Martonosi, and D.W. Clark. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2005.

[128] Xi Yang, Stephen M. Blackburn, Daniel Frampton, and Antony L. Hosking. Barriers reconsidered, friendlier still! In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, 2012.

[129] Xi Yang, Stephen M Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S McKinley. Why nothing matters: The impact of zeroing. In *Proceedings of the ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[130] Taiichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3), 1990.

[131] Wangyuan Zhang and Tao Li. Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[132] Yi Zhao, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin, and Ling Shao. Allocation wall: A limiting factor of Java applications on emerging multi-core platforms. In *Proceeding of the ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[133] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2001.