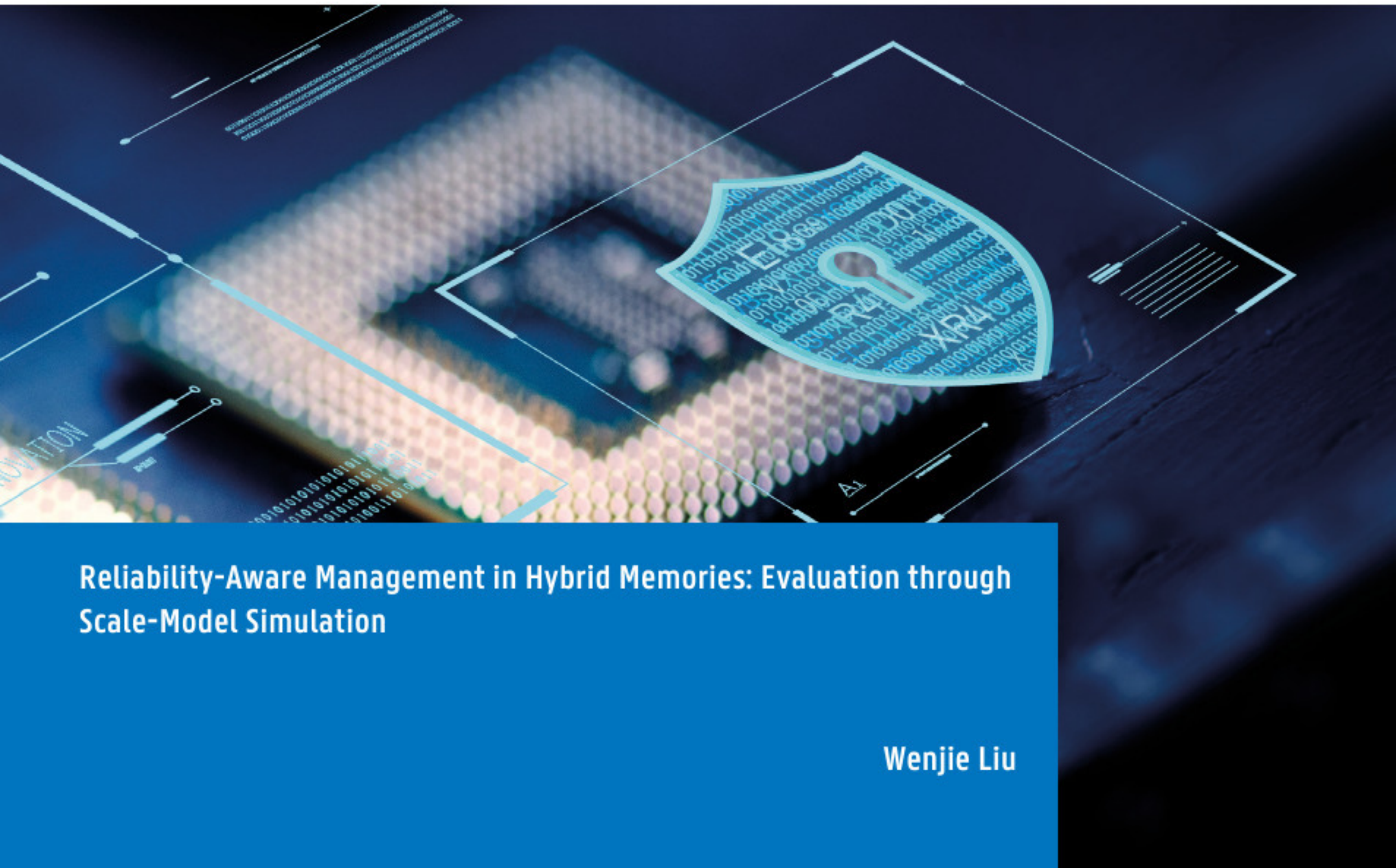
FACULTY OF ENGINEERING
AND ARCHITECTURE



Reliability-Aware Management in Hybrid Memories: Evaluation through Scale-Model Simulation

Wenjie Liu

Doctoral dissertation submitted to obtain the academic degree of
Doctor of Computer Science Engineering

Supervisor

Prof. Lieven Eeckhout, PhD

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

May 2022

GHENT
UNIVERSITY

FACULTY OF ENGINEERING AND ARCHITECTURE

Reliability-Aware Management in Hybrid Memories: Evaluation through Scale-Model Simulation

Wenjie Liu

Doctoral dissertation submitted to obtain the academic degree of
Doctor of Computer Science Engineering

Supervisor
Prof. Lieven Eeckhout, PhD

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

May 2022

GHENT UNIVERSITY

ISBN 978-94-6355-595-1
NUR 980, 987
Wettelijk depot: D/2022/10.500/36

Members of the Examination Board

Chair

Prof. Filip De Turck, PhD, Ghent University

Other members entitled to vote

Prof. Trevor E. Carlson, PhD, National University of Singapore, Singapore
Prof. Koen De Bosschere, PhD, Ghent University
Prof. Jan Fostier, PhD, Ghent University
Wim Heirman, PhD, Intel ExaScience Lab
Jennifer Sartor, PhD, Ghent University

Supervisor

Prof. Lieven Eeckhout, PhD, Ghent University

To my family

Contents

Acknowledgements

Doing a Ph.D. has been a dream since I am a master student. Reading my Ph.D. abroad, however, was not the original plan and finally became an extra gift of my life. Thanks to Professor Zhiying Wang, I began to think about the possibility of studying abroad and took my first step to work on it. Thanks to Professor Lieven Eeckhout, I got the opportunity to do research and finish my Ph.D. at Ghent University. I have delayed writing this part until the last minute because there are so many lovely people along my Ph.D. journey and I cannot thank them enough in a few words.

I want to express my deep gratitude to my advisor, Professor Lieven Eeckhout, for his valuable guidance and strong support during my Ph.D. I still remember the introductions of Lieven by my colleagues about how kind a person he is and how professional he is as a researcher. Working with him in the last four years proves that he totally deserves these compliments and he is much better. He is a hardworking and responsible advisor. He guides me how to conduct impactful research in a systematic manner. Although he is very busy, he continues to provide extensive feedback on my research, on the paper writings, and even on the presentations. Both his guidance and his work attitude inspired me and helped me become a better researcher. He is also a very caring person. I will never forget his unconditional support when I was stuck with some family issues and work pressure during the Covid outbreak. He gave me enough room to deal with my private life and provided many practical solutions to my dilemma. Without his patience, encouragement and support, I cannot imagine how to go through those tough times and finally finish my Ph.D. successfully.

Many thanks to Shoaib Akram, who played an important role in the early days of my Ph.D. life. I have been working with Shoaib in the first two years of my Ph.D., and he really gave me many suggestions on the research. He taught me to conduct valuable and interesting research instead of only focusing on paper publications. He encouraged me to insist on solving a challenging problem instead of giving up too early. His perfectionism at every step of research impressed me and inspired me to develop good research habits like him.

I want to thank all the members in my examination committee who read my thesis carefully and delivered valuable feedback despite their busy schedules. Very special thanks to Jennifer B. Sartor and Wim Heirman. I have been

working with Jennifer for my first project. She listened to my work progress patiently and gave me practical suggestions whenever I needed. I met Wim when I was stuck with some simulator issues. Wim is very professional in the field of performance modeling and I benefited a lot from his sharpness in the computer architecture design.

Many thanks to my colleagues in PerfLab: Ajeya Naithani, Yuxi Liu, Josue Feliu, Xia Zhao, Lu Wang, Shiqing Zhang, Jaime Roelandts, Almutaz Adileh, Seyyed Hossein SeyyedAghaei Rezaei, Cecilia Gonzalez-Alvarez, Sander De Pestel, Sam Van den Steen, Shoaib Akram, Kartik Lakshminarasimhan, Mahmood Naderan-Tahan, Saeideh Sheikhpour and Benyamin Eslami. I am lucky to work with them in the past four years as we spent so many happy hours in the office. I want to thank Ajeya for his help with cluster-related issues and his generous sharing of the experience on Ph.D. defense. I am very grateful to have Yuxi, Xia, Lu, and Shiqing around in Ghent. Yuxi and Xia gave me sincere suggestions on my research topics. Lu offered me considerable help to settle in Ghent. Her kindness lightened the early days of my Ph.D. life. Shiqing is always there for me, getting up early to help me test the meeting device and congratulating me on passing the internal defense. I would also like to thank the department staff for their help with administrative and technical issues. Special thanks to Marnix for the help with arranging conference travels and many thanks to Vicky and Inge for their help with various work/life related paperwork.

I thank all my friends in Ghent: Xiaodong Liu, Yun Zhou, Yuhui Wu, Sheng Yang, Xiangyu Xue, Qiming Sun, Lei Luo, Xin Cheng, Boxuan Gao, Yan Li, etc. The happy moments spending with them made my life abroad precious and unforgettable. I really cherish our friendship. I am so lucky to have Xiaodong as my roommate in the last four years. We shared ups and downs studying abroad and we supported each other whenever needed. I will never forget our heart-to-heart talks. Yun and I prepared for the Ph.D. defense at the same time, and we worked late together for the submission deadline. Her accompany relieves me a lot from the anxiety of defense preparation. I also want to thank my previous Chinese advisor in NUDT, Prof. Zhiying Wang. He encouraged me to read Ph.D. abroad and helped me apply the CSC scholarship.

Finally, I want to thank my parents for raising me and giving me endless love. Their love, wisdom, care, and strength have meant to the world to me. May good luck follow my little brother for his forthcoming GaoKao. I also want to thank my husband, Jianglong Song. If I could choose anybody at all to back me up through all the difficulties of life, it would be my husband a hundred times over. It's been ten years since we were together and thank you for always being the most reliable and supportive people to me.

Wenjie Liu
Gent, May 16, 2022

Summary

Emerging computer applications require increasing memory capacity as well as increasing bandwidth to the memory system. Unfortunately, traditional DRAM memories are constrained by the limited scalability of new chip technologies due to the decreasing reliability of individual memory cells and increasing manufacturing complexity. Consequently, computer architects must consider alternative memory technologies in order to meet the large capacity and high bandwidth requirements. Recently introduced 3D-stacked memories, such as *High-Bandwidth Memory (HBM)* in which different memory chips are placed one above the other, provide considerably higher bandwidth and lower access time at a relatively low cost. However, 3D-stacked memories have the disadvantages that capacity is limited and that reliability is inferior to conventional DRAM memories due to the higher density of memory cells. Temporary errors (e.g., *soft errors* or *transient faults*) due to cosmic rays or energy particles can lead to incorrect executions. In order to circumvent the limitations of different memory technologies, researchers have proposed hybrid memories in which, for example, a 3D-stacked memory is combined with a conventional DRAM memory. This makes it possible to offer high bandwidth (thanks to the 3D-stacked memory) and high capacity (thanks to the DRAM memory). However, the reliability of hybrid memory is limited by the weakest link, namely the 3D-stacked memory.

A concrete practical problem that arises when evaluating and exploring new architectural ideas, for example, the evaluation of hybrid memory systems, is that current simulation methodologies are inadequate. The most commonly used methodology is to model every detail of the design in a cycle-accurate manner, leading to a number of experimental problems. More specifically, the simulation time explodes when large systems have to be modeled, for example when dozens of processor cores have to be simulated for modeling a future *multicore* processor. In some cases, it is even impossible to simulate such large systems due to limitations in computing or memory capacity on the server on which the simulations are performed. However, existing techniques to solve the simulation problem, such as sampling of the execution and modeling at a higher abstraction level, do not fundamentally solve the problem and are therefore inadequate.

This doctoral thesis makes two contributions regarding the management and improvement of the reliability of hybrid memories, and two contributions regarding the simulation of large systems.

Homogeneity of Memory Allocation. Several modern programming languages use automatic memory management (e.g., *garbage collection*) to improve the productivity of the software developer and the reliability of the resulting software. Garbage collection manages memory at the level of individual objects. In this PhD thesis, we classify objects according to two criteria, namely *hotness* and *risk*, in order to map their impact on performance and reliability, respectively. Allocating hot objects in the 3D-stacked memory of a hybrid memory system offers a performance advantage (due to higher bandwidth), while allocating risky objects in the 3D-stacked memory leads to an increased vulnerability (due to less reliability). We propose to allocate objects in the 3D-stacked memory versus the conventional DRAM memory in a hybrid memory system based on the hotness and risk of those objects. Our analysis shows that the hotness and risk of an object are only weakly correlated. Consequently, it is crucial to consider both the hotness and risk. The first important contribution of this doctoral thesis consists of demonstrating that the location in the code where an object is allocated (i.e., *allocation site*) is a very accurate predictor of the hotness and risk of the object. In other words, all objects that are allocated from the same location in the program code show a similar hotness/risk profile. We refer to this property as the homogeneity of the memory allocation. We exploit this important finding in the second contribution.

Reliability-Aware Memory Management. Hybrid memory systems allow us to combine the advantages of both memory technologies and suppress the drawbacks. The advantage of the 3D-stacked memory in a hybrid memory system is the high bandwidth offered, while the disadvantages concern limited capacity and reliability. The drawbacks can be overcome by reliability-aware memory management (i.e., *reliability-aware garbage collection*). The basic idea behind reliability-aware memory management is to allocate objects in 3D-stacked memory only if those objects are hot (i.e., they are frequently accessed) and lead to a rather limited risk in terms of reliability. We propose two variants of reliability-aware memory management, namely *RiskRelief-Nursery (RR-N)* and *RiskRelief-Mature (RR-M)*. RR-N places all newly allocated objects (allocated in the so-called nursery space) in the 3D-stacked memory; older objects in the mature space that were allocated some time ago and are still reachable are placed in the DRAM memory. RR-M also makes a distinction for the mature space where hot and relatively risk-free objects are also placed in the 3D-stacked memory. Whether or not an object is hot and risky (and therefore should be placed in the 3D-stacked memory), is determined by the place where this object was allocated in the program code (first contribution of the thesis). Reliability-aware memory management in a hybrid memory system significantly improves reliability compared to a memory system that only consists of 3D-stacked memory, and at the same time improves the performance compared to a memory system only consisting of conventional DRAM memory.

More specifically, RR-N and RR-M improve the reliability by a factor of 18× and 9× compared to a 3D-stacked memory, and improve performance by 20% and 29% over a conventional memory system, respectively.

Scale-Model Simulation. To address the fundamental simulation problem of future large systems (which we also encountered in the evaluation of the second contribution), we propose a new methodology for predicting the performance of large systems based on small scale models. A scale model is in fact a miniaturized version of the large system that, because of its limited scale, can easily be simulated with existing simulation techniques. The performance obtained from the simulated scale model is then extrapolated to make a prediction for the larger system. The question arises how to construct a scale model and then extrapolate it. We show that different *shared resources*, such as the shared caches, the interconnection network and the shared memory bandwidth, should be scaled proportionally in the scale model. To extrapolate the performance of the scale model, we use machine learning (ML) techniques and we consider two variants, namely ML-based prediction and ML-based regression. ML-based prediction requires a number of simulations of the target system as input for training the ML model, while ML-based regression only requires simulations of a number of scale models (and thus not of the much larger target system). ML-based regression is therefore suitable in situations where it is impossible to simulate the target system. Predicting a multicore system with 32 processor cores based on a scale model with a single processor core results in a reduction of the simulation time by a factor of 28×. ML-based prediction and regression lead to a relative prediction error of 6.4% and 8.0%, respectively.

Scale Models for Reliability-Aware Memory Management. In the fourth and final contribution, we apply the simulation and prediction methodology based on scale models to the evaluation of reliability-aware memory management. As mentioned earlier, it was not possible to rigorously evaluate our second contribution with existing simulation methodologies because the target system is too large to simulate (32 processor cores with a hybrid memory system). Consequently, in the initial evaluation of the second contribution, we had to resort to the simulation of a proportional scale model without ML-based prediction or regression. When we fully apply the newly proposed scale model simulation methodology (the third contribution) to the evaluation of reliability-aware memory management, we conclude that the obtained performance predictions based on a proportional scale model were conservative. More specifically, we show that the performance improvement reported above for a hybrid memory system with RiskRelief memory management over a conventional memory system is an underestimation of the expected performance improvement. Based on our prediction, we expect that a hybrid memory system with RR-N and RR-M memory management improves performance by 62% (instead of 20%) and 68% (instead of 29%) over conventional memory, respectively.

Samenvatting

Hedendaagse computertoepassingen vereisen steeds meer geheugencapaciteit alsook steeds grotere bandbreedte tot het geheugensysteem. Jammer genoeg zijn traditionele DRAM-geheugens gelimiteerd door de beperkte schaalbaarheid naar nieuwe chiptechnologieën wegens een steeds afnemende betrouwbaarheid van de individuele geheugencellen en een steeds toenemende productiecomplexiteit. Bijgevolg moeten computerarchitecten alternatieve geheugentechnologieën beschouwen teneinde aan de hoge capaciteiten bandbreedtevereisten te kunnen voldoen. Recentelijk geïntroduceerde 3D-geheugens, zoals bijvoorbeeld *High-Bandwidth Memory (HBM)* waarbij verschillende geheugenchips boven elkaar geplaatst worden, leveren aanzienlijk hogere bandbreedte en lagere toegangstijd aan relatief lage kost. 3D-geheugens hebben echter als nadeel dat de capaciteit beperkt is en de betrouwbaarheid inferieur is in vergelijking met conventionele DRAM-geheugens wegens de hogere densiteit van geheugencellen. Tijdelijke fouten (Eng. *soft errors* of *transient errors*) ten gevolge van kosmische straling of energiedeeltjes kunnen leiden tot incorrecte uitvoeringen. Teneinde de beperkingen van verschillende geheugentechnologieën te omzeilen, hebben onderzoekers hybride geheugens voorgesteld waarbij bijvoorbeeld een 3D-geheugen gecombineerd wordt met een DRAM-geheugen. Dit laat toe hoge bandbreedte (dankzij het 3D-geheugen) én hoge capaciteit (dankzij het DRAM-geheugen) aan te bieden. Echter, de betrouwbaarheid van hybride geheugens is beperkt door de zwakste schakel, namelijk het 3D-geheugen.

Een concreet praktisch probleem dat zich stelt bij het evalueren en exploreren van nieuwe architecturale ideeën, zoals bijvoorbeeld de evaluatie van hybride geheugensystemen, is dat huidige simulatiemethodologieën ontoereikend zijn. De meest gebruikte methodologie bestaat erin elk detail van het ontwerp op een cyclusgetrouwe manier te modelleren, wat leidt tot een aantal experimentele problemen. Meer bepaald explodeert de simulatietijd wanneer grote systemenen gemodelleerd moeten worden, bijvoorbeeld wanneer tientallen processorkernen (Eng. *cores*) gesimuleerd moeten worden voor het modelleren van een toekomstige *multicore* processor. In sommige gevallen is het zelfs onmogelijk om dergelijke grote systemen te simuleren wegens beperkingen qua reken- of geheugencapaciteit in de server waarop de simulaties uitgevoerd worden. Bestaande technieken om het simulatieprobleem op te lossen zoals be-

monstering (Eng. *sampling*) van de uitvoering en het modelleren op een hoger abstractieniveau, bieden echter geen soelaas en zijn derhalve ontoereikend.

Deze doctoraatsthesis levert twee bijdragen m.b.t. het beheren en verbeteren van de betrouwbaarheid van hybride geheugens, en twee bijdragen m.b.t. het simuleren van grote systemen.

Homogeniteit van geheugenallocatie. Verschillende moderne programmeertalen maken gebruik van automatisch geheugenbeheer (Eng. *garbage collection*) teneinde de productiviteit van de software-ontwikkelaar en de betrouwbaarheid van de resulterende software te verbeteren. Garbage collection beheert het geheugen op het niveau van individuele objecten. In deze doctoraatsthesis classificeren we objecten aan de hand van twee criteria, namelijk belang (Eng. *hotness*) en risico (Eng. *risk*), teneinde hun impact op respectievelijk prestatie en betrouwbaarheid in kaart te brengen. Het alloceren van belangrijke objecten in het 3D-geheugen van een hybride geheugensysteem biedt een voordeel qua prestatie (wegens hogere bandbreedte), terwijl het alloceren van risicovolle objecten in het 3D-geheugen leidt een verhoogde kwetsbaarheid (wegens minder betrouwbaar). We stellen voor objecten te alloceren in het 3D-geheugen versus het conventioneel DRAM-geheugen in een hybride geheugensysteem op basis van het belang en risico van die objecten. Onze analyse toont aan dat het belang en het risico van een object slechts zwak gecorreleerd zijn. Bijgevolg is het cruciaal om zowel het belang als het risico in rekening te brengen. De eerste belangrijke bijdrage van deze doctoraatsthesis bestaat erin aan te tonen dat de plaats in de code waar een object gealloceerd wordt (Eng. *allocation site*) een zeer nauwkeurige voorspeller is van het belang en het risico van het object. M.a.w. alle objecten die gealloceerd worden vanop eenzelfde locatie in de programmacode vertonen een gelijkaardig belang/risico-profiel. We refereren naar deze eigenschap als de homogeniteit van de geheugenallocatie. Deze belangrijke vaststelling buiten we uit in de tweede bijdrage.

Betrouwbaarheidsbewust geheugenbeheer. Hybride geheugensystemen vereisen dat we de voordelen van beide geheugentechnologieën verenigen en de nadelen onderdrukken. Het voordeel van het 3D-geheugen in een hybride geheugensysteem is de hoge bandbreedte die aangeboden wordt, terwijl de nadelen beperkte capaciteit en betrouwbaarheid betreffen. De nadelen kunnen overwonnen worden door betrouwbaarheidsbewust geheugenbeheer (Eng. *reliability-aware garbage collection*). De basisidee achter betrouwbaarheidsbewust geheugenbeheer bestaat erin objecten in het 3D-geheugen te alloceren enkel en alleen als die objecten belangrijk zijn (d.i. frequent geconsulteerd worden) én tot een eerder beperkt risico leiden qua betrouwbaarheid. We stellen twee varianten van betrouwbaarheidsbewust geheugenbeheer voor, namelijk *RiskRelief-Nursery (RR-N)* en *RiskRelief-Mature (RR-M)*. RR-N plaatst alle nieuw gealloceerde objecten (die galloceerd worden in de zogenaamde *nursery* adresruimte) in het 3D-geheugen; oudere objecten in de *mature* adresruimte die een tijd geleden gealloceerd werden en nog steeds bereikbaar zijn (m.a.w. deze objecten kunnen nog steeds geconsulteerd worden), worden in het DRAM-geheugen geplaatst. RR-M maakt daarenboven nog een onderscheid voor de

mature adresruimte waarbij belangrijke en relatief risicoloze objecten eveneens in het 3D-geheugen geplaatst worden. Of een object al dan niet belangrijk en risicovol is (en dus in het 3D-geheugen geplaatst moet worden), wordt bepaald door de plaats waar dit object gealloceerd werd in de programmacode (eerste bijdrage van de thesis). Betrouwbaarheidsbewust geheugenbeheer verbetert de betrouwbaarheid aanzienlijk t.o.v. een geheugensysteem dat enkel uit 3D-geheugen zou bestaan, en verbetert tegelijkertijd de prestatie t.o.v. een geheugensysteem dat enkel uit conventioneel DRAM geheugen bestaat. Meer bepaald verbeteren RR-N en RR-M de betrouwbaarheid met een factor van respectievelijk 18× en 9× t.o.v. een 3D-geheugen. RR-N en RR-M verbeteren de prestatie met respectievelijk 20% en 29% t.o.v. een conventioneel geheugensysteem.

Simulatie van schaalmodellen. Teneinde het fundamentele probleem van de simulatie van toekomstige grote systemen aan te pakken (waar we eveneens op gestoten zijn bij de evaluatie van de tweede bijdrage), stellen we een nieuwe methodologie voor voor het voorspellen van de prestatie van grote systemen op basis van kleine schaalmodellen. Een schaalmodel is in feite een geminiaturiseerde versie van het grote systeem dat wegens zijn beperkte schaal eenvoudig te simuleren valt met bestaande simulatietechnieken. De prestatie bekomen op basis van het gesimuleerde schaalmodel wordt vervolgens geëxtrapoleerd teneinde een voorspelling te maken voor het groter systeem. De vraag stelt zich hoe een schaalmodel te construeren en vervolgens te extrapoleren. We tonen aan dat de verschillende gedeelde componenten (Eng. *shared resources*), zoals de gemeenschappelijk caches, het interconnectienetwerk en de gedeelde geheugenbandbreedte, best proportioneel geschaald worden in het schaalmodel. Voor het extrapoleren van de prestatie van het schaalmodel maken we gebruik van machine learning (ML), en we beschouwen twee varianten, namelijk ML-gebaseerde voorspelling en ML-gebaseerde regressie. ML-gebaseerde voorspelling vereist een aantal simulaties van het doelsysteem als input voor het trainen van het ML-model, terwijl ML-gebaseerde regressie enkel simulaties vereist van een aantal schaalmodellen (en dus niet van het veel groter doelsysteem). ML-gebaseerde regressie is bijgevolg geschikt in situaties waarbij het onmogelijk is het doelsysteem te simuleren. Het voorspellen van een multicore systeem met 32 processorkernen op basis van een schaalmodel met een enkele processorkern levert een reductie van de simulatietijd op van een factor 28×. ML-gebaseerde voorspelling en regressie leiden tot een relatieve voorspellingsfout van respectievelijk 6.4% en 8.0%.

Schaalmodellen voor betrouwbaarheidsbewust geheugenbeheer. In de vierde en laatste bijdrage passen we de simulatie- en voorspellingsmethodologie op basis van schaalmodellen toe op het evalueren van betrouwbaarheidsbewust geheugenbeheer. Zoals eerder aangehaald was het niet mogelijk om onze tweede bijdrage rigoureus te evalueren met bestaande simulatiemethodologieën wegens de te grote schaal van het doelsysteem (32 processorkernen met hybride geheugensysteem). Bijgevolg hebben we bij de initiële evaluatie van de tweede bijdrage onze toevlucht moeten nemen tot de simulatie van

een proportioneel schaalmodel zonder ML-gebaseerde voorspelling of regressie. Wanneer we de nieuw voorgestelde simulatie- en voorspellingsmethodologie op basis van schaalmodellen (derde bijdrage) ten volle toepassen voor de evaluatie van betrouwbaarheidsbewust geheugenbeheer, concluderen we dat de bekomen prestatievoorspellingen op basis van een proportioneel schaalmodel conservatief waren. Meer specifiek tonen we aan dat de hierboven gerapporteerde prestatieverbetering voor een hybride geheugensysteem met RiskRelief-geheugenbeheer t.o.v. een conventioneel geheugensysteem een onderschatting is van de te verwachten prestatieverbetering. Op basis van onze voorspelling verwachten we dat een hybride geheugensysteem met RR-N en RR-M geheugenbeheer de prestatie verbeteren met respectievelijk 62% (i.p.v. 20%) en 68% (i.p.v. 29%) ten opzichte van een conventioneel geheugen.

List of Figures

List of Tables

List of Abbreviations

ACE	Architecturally Correct Execution
AI	Artificial Intelligence
AVF	Architectural Vulnerability Factor
CPI	Cycles Per Instruction
CPU	Central Processing Unit
CSL	Cross-Section Link
DRAM	Dynamic Random Access Memory
DT	Decision Tree
ECC	Error Correction Code
FeRAM	Ferroelectric Random-Access Memory
FIT	Failure In Time
FMID	Fixed-Midpoint
GB	Giga Byte
GC	Garbage Collection
GP	General Purpose
HBM	High-Bandwidth Memory
HMC	Hybrid Memory Cube
IPC	Instructions Per Cycle
ISA	Instruction-Set Architecture
JIT	Just-In-Time
JVM	Java Virtual Machine

KB	Kilo Byte
L1I	Level-1 Instruction Cache
L1D	Level-1 Data Cache
L2	Level-2 Cache
L3	Level-3 Cache
LLC	Last-Level Cache
LOS	Large Object Space
MB	Mega Byte
MC	Memory Controller
ML	Machine Learning
MPKI	Misses Per Kilo Instructions
MRAM	Magneto-Resistive Random-Access Memory
MRAT	Moving-Ratio
MTTF	Mean Time To Failure
MTBF	Mean Time Between Failures
NAND	NOT-AND
NoC	Network-on-Chip
NRS	No Resource Scaling
NUCA	Non-Uniform Cache Architectures
NUMA	Non-Uniform Memory Access
NVM	Non-Volatile Memory
OS	Operating System
PC	Program Counter
PCM	Phase Change Memory
PRS	Proportional Resource Scaling
ReRAM	Resistive Random-Access Memory
RF	Random Forest
ROB	Re-Order Buffer

RR RiskRelief

RR-N RiskRelief-Nursery

RR-M RiskRelief-Mature

SER Soft Error Rate

SRAM Static Random Access Memory

STP System Throughput

STT Spin-Transfer Torque

SVM Support Vector Machines

TLB Translation Lookaside Buffer

TSV Through-Silicon Via

Chapter 1

Introduction

1.1 Motivation

In this section, we first focus on the challenges related to managing emerging hybrid memory systems and then motivate the need for novel simulation and prediction techniques for exploring and evaluating future large-scale systems.

1.1.1 Soft Error Reliability in Hybrid Memory Systems

High-bandwidth memory (HBM) is very popular in emerging hybrid memory designs as it satisfies the ever-evolving bandwidth requirements of new throughput-oriented compute platforms. HBM delivers 4–8× higher bandwidth than traditional DRAM memory using 3D die-stacking. Unfortunately, HBM is limited in capacity and has a high soft error rate due to high bit density and new failure modes [84, 130]. Combining HBM and DRAM into a hybrid memory system can meet the need for high capacity provided by DRAM and benefit from high bandwidth provided by HBM. The reliability of hybrid memory systems, however, is still a concern with no proper management, especially for the HBM partition.

Hybrid memory systems are typically managed through hardware and operating system (OS) approaches. For example, HBM is organized as a cache for conventional DRAM memory in hardware approaches [43, 44, 89, 96, 107] and OS solutions map frequently accessed pages to HBM [135, 145, 146, 165]. The aforementioned proposals intensively focus on improving performance for hybrid memories, leaving the reliability problem as an open question. Soft error rates in production systems are continuously increasing, and they grow proportionally with information density [98]. Researchers have recently refocused their attention on addressing the low reliability of emerging memory systems but faced new challenges with proposals based on existing techniques. Specif-

ically, it is insufficient to tackle the reliability problem using hardware-only approaches because they will require impractical error detection and correction capabilities [117]. The OS-based approach leverages performance and reliability for a heterogeneous memory architecture but it operates at a coarse-grained page granularity and frequent page migrations incur significant performance penalties [69].

Fortunately, garbage collection in managed programming languages provides a novel insight to manage data in hybrid memory systems. In this thesis, we focus on how to improve the reliability of hybrid HBM-DRAM systems while delivering high performance and memory capacity. The subchallenges include: (1) how to quantify the performance and reliability characteristics of applications executing on the target system, (2) how to predict the hotness and risk of allocated data to guide data management, and (3) how to design a data management policy and organize hybrid memory partitions.

1.1.2 Large-Scale System Simulation

Computer architects extensively rely on simulation to steer future processor research and development. Simulating architecture and predicting performance for a future computer system is a critical and challenging problem. Considerable approaches have been proposed to tackle this challenge from the perspective of either system simulation or performance modeling.

The traditional approach is to deploy detailed architectural simulations such as cycle-accurate and cycle-level simulation. However, simulating every detail of the target system increases the simulation complexity and incurs extremely high simulation time overheads. Scaling down workloads speeds up the simulation by selecting representative regions of an application to execute on the system and extrapolating the evaluated results to the whole execution process. This method saves simulation time to some extent but leaves a challenging problem for system simulation – that is, simulation infrastructures may not support simulating a large-scale computer system because of infrastructure limitations or insufficient compute and memory capacities in the simulation host system.

Performance modeling is an alternative approach to model interactions in a designed processor. ML-based techniques first train prediction models using simulation results obtained from detailed simulations and then evaluate performance or other metrics for the target system through prediction models. The key challenges of ML-based models are: (1) it is time-consuming to obtain sufficient and representative training samples from detailed simulations, and (2) such models provide limited insight into the evaluated system by treating the system as a black box. Analytical models, on the other hand, use mathematical formulas to model application behavior executing on a designed system based on simplifying assumptions and first principles. Application profiling is a one-time cost and the performance estimation is quite fast because it only contains a set of mathematical equations, making it suitable for a fast, early-

stage architecture exploration. However, the lack of modeling overlap effects and tracking timing-sensitive behavior makes analytical models infeasible to simulate increasingly large systems.

In this thesis, we target efficient and accurate architectural simulation for future large-scale systems. Specifically, we combine architectural simulation with machine learning techniques to predict performance for large-scale systems based on detailed simulation of a scaled-down configuration of the target system. The key challenges involve how to construct representative scale models for the target system and how to build an accurate extrapolation model based on the scale model predictions.

1.1.3 Managed Language Simulation

Multiprogrammed managed language workloads have received considerable attention from computer architects due to the emerging fields of cloud computing and micro-services [12, 67, 80, 120, 168]. Fast and accurate architectural simulation of multiprogrammed managed language workloads is thus becoming an increasingly critical problem.

Managed programming languages, such as Java, Python or JavaScript, expose a higher level of abstraction to the programmer than native programming languages like C and C++. Higher abstraction characteristics of managed programming languages incur much more memory allocation [5, 194], which stresses the memory subsystem and further increases simulation time compared to the simulation of workloads written in native programming languages. Another reason for the high simulation time of managed language workloads is the presence of a runtime environment. For example, the Java Virtual Machine (JVM), a widely-used language runtime, provides bytecode interpretation, just-in-time compilation, and garbage collection to facilitate faster development times and to provide platform independence. These services run in their own context, and increase the overall execution and simulation time [2, 33, 132]. As a result, simulating a Java workload requires simulation of the entire application, including the overhead introduced by the just-in-time compiler and the garbage collector. A single-instance Java workload that we simulate executes up to 30 billion instructions and multiprogramming undoubtedly increases the simulation overhead due to contention in shared resources. A final factor that inhibits fast simulation of multiprogrammed managed language workloads is the internal synchronization of parallel simulators. Simulators, such as Sniper [36], provide high-speed, parallel simulation for multithreaded and multiprogrammed workloads. To keep the simulation correct and accurate, synchronization of cores and shared resources is required, which inhibits the scalability of simulation time for multiprogrammed managed language workloads. All these challenges motivate our newly proposed scale-model architectural simulation methodology for simulating a large-scale system with multiprogrammed managed language workloads executing on it.

1.2 Key Contributions

This thesis makes four major contributions.

Contribution #1: Allocation-Site Homogeneity

Garbage collection (GC) offered by managed programming languages automatically manages virtual heap memory and relieves the programmer from performing manual memory management where the programmer specifies how to allocate and deallocate objects. More specifically, generational GCs place newly allocated objects in a small nursery space and copy nursery survivors to a large mature space during a nursery collection. To address the performance and reliability challenges for a hybrid HBM-DRAM memory, we need to profile the application execution on the target system. We build upon two notions, namely hotness and risk, to quantify the performance and reliability characteristics of the evaluated applications. Intuitively, hotness refers to how frequently an object is accessed and risk refers to how susceptible an object is to soft errors. The simple placement of hot objects in HBM improves performance, while significantly hurting the reliability of the overall system. The first insight is to place objects in HBM versus DRAM based on their hotness and risk to benefit from both memory components – that is, delivering high reliability while achieving high performance.

We start by quantifying the distribution of object hotness and risk for several representative benchmarks. We observe that a large fraction of mature-object accesses are captured by a relatively small fraction of the mature heap. For benchmark Fop for example, 90% of the mature-object accesses are concentrated to only 32% of the mature heap. This observation suggests an opportunity to allocate a small fraction of hot objects in HBM to improve performance while placing the bulk of the mature heap in DRAM to exploit its capacity. On the risk side, we report the distribution of relatively hot objects and observe a remarkable variation in risk, implying that hotness is not predictive for risk. In other words, object hotness and risk are weakly correlated. Therefore, we need a method to predict and classify objects for both hotness and risk combined.

The second key insight is that allocation site is an accurate predictor for both object hotness and risk. To demonstrate this, we first compute the hotness and risk for all objects and then compute the fraction of hotness and risk across objects for each allocation site. We define *homogeneity* of an allocation site with respect to hotness, risk or combined hotness/risk, as the fraction of objects that are classified in the same category. For example, for the combined hotness and risk metric, perfect (100%) homogeneity means that all objects allocated from a certain site are both hot and low-risk, or they are not, i.e., they are either cold or high-risk. We report the heap volume distribution over allocation-site homogeneity for hotness, risk and the combined hotness/risk and make two observations. First, heap volume increases with decreasing allocation site homogeneity. For example, a relatively small fraction of the total heap volume is covered at 100% homogeneity and the entire heap is covered at 50%

homogeneity. Second, the combined metric outperforms the isolated hotness and risk metrics. For example, for 90% homogeneity, more than 97% of the heap is correctly classified for the combined metric and the percentage numbers are 79% and 72% for hotness and risk, respectively. This implies that allocation site is a more accurate predictor for hotness and risk combined, than for hotness and risk in isolation. Based on the two discussed observations, we conclude that the allocation site is a very accurate predictor for object hotness and risk, which enables the proposed garbage collection assisted with allocation-site prediction.

Contribution #2: Reliability-Aware Garbage Collection

We propose two *reliability-aware garbage collectors* for hybrid HBM-DRAM memory to minimize the soft error rate while maximizing the overall performance of the application. These collectors place hot and low-risk objects in HBM memory to improve reliability and performance, and place the remaining objects in DRAM memory to exploit its large capacity. Specifically, RiskRelief-Nursery (RR-N) places the nursery space in HBM and the rest, such as the mature space and the large object space, in DRAM. It requires minimal changes to Java runtime but is highly effective in delivering low soft error rates compared to an HBM-Only system and improving performance compared to a DRAM-Only system. RiskRelief-Mature (RR-M) also places newly allocated objects in the nursery space in HBM, and copies hot and low-risk nursery survivors to the HBM mature space instead of DRAM mature space during the nursery collection for a larger performance improvement.

We observe that mature object hotness and risk are predictable on a per allocation-site basis. Based on this observation, we propose a heuristic to classify allocation sites as DRAM and HBM. Allocation sites are classified as HBM if most of the objects they allocate are hot and low-risk. All other allocation sites default as DRAM. We generate this per allocation-site advice offline and feed it to RR-M. In turn, RR-M uses the advice during runtime to place nursery survivors in HBM or DRAM. Our proposed heuristics expose previously unseen Pareto-optimal trade-offs between execution time and soft error rate. A single profiling run generates a range of advice files for the GC runtime. Thus, depending upon factors such as environmental conditions, available HBM capacity and performance goals, a system operator can adjust the advice fed to RR-M to meet specific demands, such as exploiting the rich trade-offs between performance, SER and memory capacity.

Our experimental results show that RR-N reduces the overall soft error rate by 18× on average compared to an HBM-Only system, while improving the performance over a homogeneous DRAM-Only system by 20%. The state-of-the-art OS solution by Gupta et al. [69] achieves similar SER as RR-N, however, performance is substantially worse (even worse than the DRAM-Only system) due to the high cost of TLB shootdowns on modern x86 multicores [135]. Both RR-N and the prior OS approach use a modest 128 MB of HBM on a 32-core platform. RR-M uses an additional 18% of HBM capacity but delivers 29%

higher performance compared to a DRAM-Only system. Higher HBM capacity affects the overall SER, and RR-M reduces SER by 9× over HBM-Only.

The above two contributions are published in:

W. Liu, S. Akram, J. B. Sartor, and L. Eeckhout. Reliability-Aware Garbage Collection for Hybrid HBM-DRAM Memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(1):1–25, 2021

Contribution #3: Scale-Model Architectural Simulation

We propose *scale-model simulation*, a novel methodology to predict performance for future large-scale multicore systems. Scale-model simulation combines architectural simulation with machine learning techniques to predict performance for future systems based on a detailed simulation of a scaled-down configuration of the target system, called the *scale model*. Scale-model simulation first simulates a scale model of the target system. Performance for the target system is then predicted through extrapolation. Scale-model architectural simulation involves two key concerns: (1) how to construct scale models and (2) how to build an accurate extrapolation model based on the scale-model predictions. For the first objective, the challenge when constructing scale models for general-purpose multicore processors is how to deal with shared resources. One option is to simply scale the number of cores in the scale model while keeping the shared resources unchanged as in the target system – we refer to this approach as *No Resource Scaling (NRS)*. We find for our suite of SPEC CPU2017 workloads that not scaling shared resources leads to largely inaccurate scale models with an average 60% prediction error (and up to 94%) for a single-core scale model versus a 32-core target system. The alternative option is to proportionally scale the shared resources – we refer to this approach as *Proportional Resource Scaling (PRS)*. More specifically, when scaling the number of cores by a factor F in the scale model relative to the target system, the shared resources are also reduced proportionally by the same factor, i.e., LLC capacity, NoC bisection bandwidth and memory bandwidth are reduced by a factor F. We find that proportional resource scaling delivers substantially more accurate scale models, with an average prediction error of 14.7% and at most 32.2% for a single-core scale model relative to a 32-core target system.

After constructing scale models for the target system, we need to explore extrapolation techniques to yield much more accurate performance predictions for the target system based on the performance results obtained from scale models. We propose and evaluate two extrapolation methods that leverage Machine Learning (ML) to infer prediction models that predict target-system performance based on scale-model measurements, namely ML-based prediction and ML-based regression. The key difference between both methods is that ML-based regression does not require simulation runs of the target system during training, in contrast to ML-based prediction. This has important implications in practice. We have to resort to ML-based regression if it is impossible to simulate the target system for some reason (e.g., too long simulation time or other infrastructure-related limitations). We explore a variety of machine

learning techniques, including decision trees, random forest and support vector machines (SVM) in the context of scale-model simulation, and we find that SVM is the most accurate. In addition, we evaluate a number of regression-based extrapolation methods (i.e., linear, power and logarithmic) and find that logarithmic regression is the most accurate.

Our evaluation using multiprogram SPEC CPU2017 workloads demonstrates high accuracy of the scale-model simulation. Considering a single-core scale model and a 32-core target system, we report that for homogeneous multiprogram workload mixes, SVM-based prediction yields an average prediction error of 6.4% (and 20.8% max error). SVM-based regression is slightly less accurate as it does not involve simulations of the target system when training the prediction model. SVM-based regression yields an average prediction error of 8.0% (and 26.4% at most). Scale-model simulation leads to substantial simulation speedups. Training the prediction model is a one-time cost that can be amortized across many predictions. Once the prediction model has been trained, scale-model simulation is fast. It only requires running a simulation of the application of interest on the single-core scale model, which is substantially faster than running a simulation of the target system, i.e., in our experimental setup in which we use Sniper [36] on a high-end 36-core Intel PowerEdge R440 server, we find that simulating a single-core scale model is 28× faster than simulating the 32-core target system.

This contribution is published in:

W. Liu, W. Heirman, S. Eyerman, S. Akram, and L. Eeckhout. Scale-Model Simulation. *IEEE Computer Architecture Letters (CAL)*, 20(2):175–178, 2021

An extended version of this work is published in:

W. Liu, W. Heirman, S. Eyerman, S. Akram, and L. Eeckhout. Scale-Model Architectural Simulation. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022, Accepted

Contribution #4: Scale-Model Simulation for Reliability-Aware Garbage Collection

Scale-model architectural simulation was proven to be a fast and accurate simulation methodology for large-scale systems in the third contribution, which encourages an optimized simulation of reliability-aware memory management proposed in the second contribution. More specifically, a rigorous evaluation of our proposed reliability-aware garbage collection was inhibited because the designed system was too large to simulate. We find that simulating a single 16-instance Java workload on a 16-core processor takes more than two weeks on average and over a month at most. The evaluation of a 32-instance workload execution on a 32-core system was even impossible due to the limited computing and memory capacity in the simulation host system. As a result, we had to evaluate the proposed reliability-aware memory management in a hybrid memory system with a scaled single-core model without ML-based extrapolation in the second contribution.

In this fourth contribution, we apply the simulation and prediction methodology based on scale models to the evaluation of reliability-aware garbage collection. We conclude that the performance improvement obtained based on a scaled single-core model (reported in the second contribution) is conservative – that is, the performance improvement obtained from the RiskRelief collectors is expected to be even higher with a more accurate and elaborate evaluation. The experimental results show that RR-N yields an average performance benefit of 20% compared to a DRAM-only system based on the single-core simulations and 62% based on the 32-core predictions using ML-based regression. RR-M improves performance by 29% on average over DRAM-only using single-core simulations versus 68% using 32-core predictions with ML-based regression.

1.3 Structure and Overview

This dissertation is organized into six chapters.

Chapter 2 describes recent trends in memory system development, introduces necessary background on system reliability, and discusses reliability estimation techniques for emerging hybrid memory systems. It also introduces representative simulation and prediction methodologies for modern computer processors. Finally, this chapter explains how to evaluate Java workloads using state-of-the-art techniques to ease the understanding of the experimental setups in Chapters 3 and 5.

In Chapter 3, we first demonstrate the performance and reliability challenges for the current memory system and then explore allocation sites to predict the hotness and risk of objects. Finally, we propose reliability-aware memory management (i.e., reliability-aware garbage collection) for hybrid HBM-DRAM memories. We evaluate the proposed garbage collectors with simulation results from the simulator and emulation results for the real hardware. The experimental results show that the proposed garbage collection can improve system reliability while maximizing overall application performance. It also manages hybrid HBM-DRAM memory significantly better than the state-of-the-art OS approach.

Chapter 4 targets the performance prediction for future large-scale computer systems. We propose a scale-model architectural simulation to predict performance for the target system based on the simulation results of scale model methodology. This chapter presents how to construct scale models and how to extrapolate performance results obtained from scale models to predict performance for a large target system. Two machine learning-based extrapolation techniques, namely ML-based prediction and ML-based regression, are proposed and evaluated using homogeneous and heterogeneous workload mixes. We also evaluate the prediction accuracy, simulation speedups and sensitivity of the proposed scale-model simulation technology.

Chapter 5 first illustrates the challenges in simulating multi-programmed Java workloads on a large-scale system. It then introduces the application of scale-model simulation (proposed in Chapter 4) to the evaluation of the target system (which is proposed in Chapter 3 and is too large to simulate with existing simulation techniques). We evaluate the simulation speedup achieved from scale-model simulation, the performance prediction accuracy of the proposed approach on multiprogrammed Java workloads executing on the target system, and the feasibility of prototyping a future system that may be prohibitive to be simulated due to infrastructure limitations and/or insufficient memory and computing capability. We also verify a critical observation obtained from Chapter 3 that the performance predictions based on small-scale models are representative and conservative compared to those results expected from the target system.

Finally, in Chapter 6 we conclude the dissertation and discuss some potential avenues for future work.

Chapter 2

Background

In this chapter, we present the background on recent trends in memory system design, basic knowledge of system reliability, and existing simulation and prediction methodologies. Section 2.1 discusses the challenges for DRAM scaling, illustrates the development of emerging memory technologies, and briefly introduces the assumed memory architecture used in this dissertation. Section 2.2 introduces the basic knowledge related to system reliability. We first explain terminology in the fault tolerance domain. We then introduce the evaluation metrics to quantify the reliability of a system and we describe Architecture Vulnerability Factor (AVF) analysis, an estimation methodology used for assessing soft error reliability in this dissertation. Section 2.3 and Section 2.4 summarize the existing simulation and modeling techniques to simulate the (micro-)architecture and predict performance for a designed system. Finally in Section 2.5, we describe the current state-of-the-art in benchmarking Java workloads.

2.1 Memory System Trends

2.1.1 DRAM Challenges

Main memory is a critical component of a modern computer system. Dynamic random access memory (DRAM) has served as the main memory over the past decades and is experiencing difficult architecture and technology scaling problems derived from recent system design, technology trends and emerging applications. On the architecture side, an increasing number of processing cores [45, 93, 173] (e.g., multi-core homogeneous/heterogeneous processors, diverse accelerators and graphic processing units) are building upon the memory system and delivering rapacious demand for memory bandwidth, system reliability, power consumption, etc. [125, 128, 172] On the technology side, emerging

memory structures/technologies launch a huge attack on well-established memory management policies. Simply implementing current DRAM-based memory techniques on future memory architectures is usually incompatible and insufficient to achieve the best of new techniques. Several data placement techniques [131, 179, 180] have been proposed to improve performance, reliability and/or mitigate power overhead for heterogeneous architectures. Finally, on the application side, applications executing on the cores are becoming increasingly data and memory intensive, which requires efficient manipulation of large amounts of data. For example, big data analytics uses advanced analytical techniques [50, 61, 77, 187] to uncover information, such as hidden patterns, unknown correlations and market trends, from very large diverse data sets. It can be used for better decision making, preventing fraudulent activities, among other things. One predominant challenge for such techniques is the processing of massive amounts of data, which overwhelms the memory capacity and bandwidth of today's computer systems.

Putting it all together, emerging architecture, technology and application trends exacerbate challenges for the scaling of conventional DRAM memory. Techniques that leverage the advantages of multiple memory levels and pursue cooperation across different computing layers appear to be promising and crucial for solving problems related to performance, reliability, capacity and energy efficiency for the memory or the whole system.

2.1.2 Emerging Memory Technologies

Emerging big data and artificial intelligence (AI) techniques, including machine learning, drive innovations across academic and industry fields, accompanied with new memory technologies. Computer architects devote much effort to alternative memory technologies that may replace DRAM, focusing on either non-volatile or volatile memories. Another direction, instead of replacing DRAM, is to combine competitive options with DRAM and devise appropriate approaches to exploit the advantages of both worlds.

Non-volatile memories are extensively investigated and highly expected to replace existing memories. Compared to DRAM-based main memory technologies, they are usually more reliable, easily programmable, and promise better scalability. Emerging non-volatile memories include Magneto-Resistive Random-Access Memory (MRAM), Ferroelectric Random-Access Memory (FeRAM), Phase Change Memory (PCM), Resistive Random-Access Memory (ReRAM), etc. MRAM [32, 63], implied by its name, stores the data using magnetic storage elements. The storage elements are formed by two ferromagnetic plates, each of which can hold a magnetic field, separated by a thin insulating layer. One of the two plates is a permanent magnet set to a particular polarity; the other's field can be changed to match that of an external field to store memory. Spin-transfer torque random-access memory (STT-RAM) is an advanced MRAM but with better scalability over traditional MRAM. The spin-transfer

torque (STT) is an effect in which the orientation of a magnetic layer in a magnetic tunnel junction or spin valve can be modified using a spinpolarized current. Spin-transfer torque technology has the potential to meet low current requirements and reduce memory cost; however, the amount of current needed to reorient the magnetization is currently too high for most commercial applications. PCM utilizes the unique behavior of chalcogenide glass whereby the heat produced by the passage of an electric current switches this material between two states – the amorphous and the crystalline state. The different states have different electrical resistance which can be used to store data. The advantages of PCM make it one of the most promising technologies for being an alternative to existing memory. For example, PCM is byte-addressable, persistent and can offer more capacity than DRAM. However, challenges, such as high latency and low write endurance, still exist, preventing it from being widely deployed. ReRAM is a nonvolatile memory similar to PCM. The technology concept is that a dielectric, which is normally insulating, can be made to conduct through a filament or conduction path formed after the application of a sufficiently high voltage. Arguably, this is a memristor technology and should be considered as a potentially strong candidate to challenge NAND Flash.

Volatile memory, DRAM specifically, has been the predominant physical substrate for implementing main memory. Introducing the 3D die-stacking technique to conventional DRAM opens up a promising avenue to mitigate the growing demands of increased bandwidth and low power consumption at a relatively low cost. Hybrid Memory Cube (HMC) [140] is a high-performance RAM interface designed for 3D die-stacked DRAM memory. HMC combines through-silicon vias (TSVs) and microbumps to connect multiple dies of memory cell arrays on top of each other. One of the major goals of HMC is to eliminate the duplicative control logic of modern DIMMs. In the process streamline design, HMC links the entire stack in a 3D configuration and then uses one control logic layer to cater to all traffic. HMC is explicitly designed to respond to multi-core scenarios and deliver data with substantially higher bandwidth and lower overall latency. High bandwidth memory (HBM) [86], another 3D counterpart to DRAM, is an innovative memory technology which stacks multiple DRAM layers vertically, where layers are also connected by TSVs. HBM offers more channels per device, smaller page sizes per bank, wider activation windows and a dual command line for simultaneous read and write. These features distinguish HBM to provide performance and power improvements in case of bandwidth-sensitive workloads.

Memory technologies discussed above target increasing memory requirements from different perspectives. However, no existing or emerging single memory module can provide the lowest latency, highest bandwidth, largest capacity, highest reliability, and lowest power consumption at the same time. Therefore, homogeneous memory systems are often not sufficient for the upcoming computing era with big data, artificial intelligence, IoT, cloud, etc. Using combined DRAM technologies for main memory systems to improve energy efficiency on traditional CPUs has been explored by several

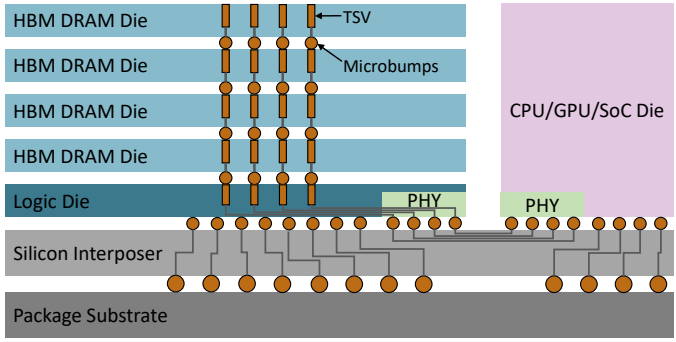


Figure 2.1: HBM architecture: HBM vertically stacks multiple DRAM dies which are interconnected by microscopic wires called through-silicon vias (TSVs).

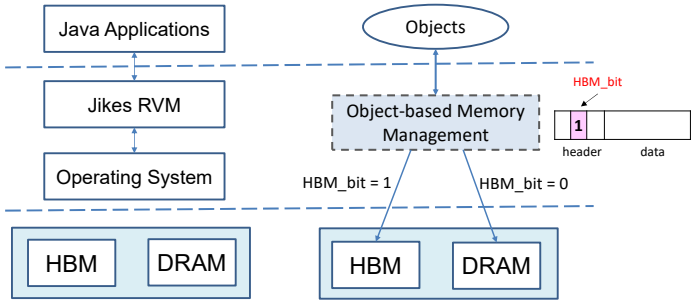


Figure 2.2: Assumed memory and the computer system level hierarchy.

groups [22, 38, 100, 122, 139, 143, 147]. Such work has focused on overcoming the performance limitations that future non-volatile memory technologies may have compared to existing DRAM designs. In addition to off-package memories, upcoming on-package memories provide opportunities for latency reduction by either increasing the number of banks available to the application [51] or balancing the bandwidth with power consumption [193]. An alternative to treating heterogeneous memory systems as a flat memory space is to treat one technology as a cache for the other [92, 116]. Although this cache-oriented design has the advantage of being transparent to the programmer, OS and runtimes, few implementations [164] take advantage of the additional bandwidth when employing heterogeneous memory.

2.1.3 Hybrid HBM-DRAM Memory System

Die-stacked DRAM is a technology that has recently been integrated into high-performance systems. High bandwidth memory (HBM), a new 3D die-stacked memory, has been explored as one of the most promising memory technologies. Figure 2.1 illustrates a basic architecture of HBM [13]. The fundamental components of HBM consist of multiple DRAM dies stacked one above the other and an optional base die at the bottom. DRAM dies are vertically stacked and interconnected by through-silicon vias (TSVs) and microbumps. The base die provides I/O buffers and test logic. HBM is often connected to the memory controller on a GPU or CPU through a substrate, such as a silicon interposer. Alternatively, the memory die could be stacked directly on the CPU or GPU chip.

HBM can be integrated into the storage hierarchy in different ways due to variable optimization targets. Some studies have focused on hardware caching techniques [43, 70, 191] to make use of the stacked memory, but these approaches require complex hardware changes and cannot leverage the stacked memory to increase the overall memory capacity of the system. Recent trends [44, 69] feature stacked memory next to traditional memory, making them work side by side to obtain a hybrid or heterogeneous memory system. The ensuing challenge is to figure out what the limitations and constraints are, and how applications can make efficient use of new high-bandwidth memory and the traditional memory to achieve best of both worlds.

In this thesis, we assume a hybrid HBM-DRAM memory system that combines on-package HBM with off-package conventional DRAM memory – similar to the memory system equipped on the Intel Knights Landing (KNL) processor [169]. Specifically, HBM is integrated on-package while DRAM is off-package and connected by DDR4 channels. Figure 2.2 demonstrates our assumed memory system and the computer system level hierarchy we used. We present HBM as an additional NUMA node to the OS to exploit full memory capacity. We modify the memory management toolkit (MMTk) of Java Virtual Machine (JVM) – Jikes' MMTk in our case – to split the virtual heap into HBM and DRAM partitions. The key observations of Java applications executing on such a system motivate us to propose a new class of garbage collectors to manage our hybrid memory at the object level while requiring no extra hardware support. The assumed HBM-DRAM memory system will be built and evaluated in Chapters 3 and 5. We will introduce the detailed proposals in Chapter 3.

2.2 System Reliability

We now introduce some basics about system reliability.

2.2.1 Terminology

Reliability describes the capability of a system to continuously deliver expected services. It is formally defined as the probability that a system will produce outputs without failure up to some given time t [114]. The definition of reliability presents its correlation with *failure* which is a possible output of an *error* caused by a hardware or software *fault*. We now discuss the definitions of a fault, error and failure, along with their cause-and-effect relationships.

A *fault* is the physical defect or temporal malfunction from the hardware perspective. For example, it can be caused by a flaw in the manufacturing process of a silicon chip or undesirable changes in temperature. Programming bugs, derived from incorrect specification or human mistakes, may lead to unexpected errors and then defined as software faults. A fault in a device is permanent if it is unrecoverable. For CMOS technology they can be classified as extrinsic and intrinsic faults. Extrinsic faults are caused during device manufacturing by contamination or burn-in testing. Intrinsic faults are directly related to the CMOS ageing effects, where the performance of device degrades over time. Permanent faults reappear upon every use of the device. Transient faults occur when energy particles, such as alpha particles, cosmic rays or thermal neutrons, strike the transistors or the logical gates. Such faults are random and no longer present when the driving source disappears.

An *error* is a deviation from the required operation of a system or subsystem, which is caused by a fault. For example, an error occurs when the sequential logic of a circuit (e.g., register files or pipeline registers) generates an unexpected value. An error makes the fault apparent, whereas not all faults lead to errors. Masking mechanisms such as electrical masking, logic masking and timing masking [9, 134] could prevent faults from forming errors. The error caused by a permanent fault is called a permanent error or a hard error. Hard errors need a fix at the hardware level or they are entirely unrecoverable, which reaches out of our research. A soft error, on the other hand, stems from a transient fault and causes a temporary unintended condition in the device. Our research in this thesis mainly focuses on the analysis and tolerance of soft errors caused by transient faults.

A *failure* indicates the occurrence of unanticipated behavior of a system – that is, a possible outcome of an error. Unexpected behavior could be generating wrong outputs for a program execution or storing mismatched values in memory. Not all errors lead to a failure. For example, failures will be avoided if erroneous values in register files are overwritten before being stored in memory, or the subroutine is not called, making its programming bugs invisible. Failures can be avoided or reduced through detecting and correcting faults or errors in advance; we discuss the detailed techniques in the next subsection.

Minimum Hamming Distance	1	2	3	3	4	4	5	5	5
Number of bit errors which can be detected	0	1	1	2	2	3	2	3	4
Number of bit errors which can be corrected	0	0	1	0	1	0	2	1	0

Table 2.1: Relationship between minimum Hamming distance and number of bit errors that can be detected and corrected.

2.2.2 Fault-Tolerant Techniques for Memory

Information redundancy and coding techniques have been widely used to protect computer systems. Coding techniques can be used to detect and/or correct single-bit or multibit errors. Examples of common error codes used in computer systems include some fundamental schemes such as parity codes, single error correction double error detection (SECDED) codes and cyclic redundancy check (CRC) codes, as well as some advanced versions like Chipkill and parity prediction circuits. We mainly introduce SECDED codes and Chipkill-based protection in this section, as they will be adopted in this thesis for the following proposals.

The basic idea of error codes is to use a *code word* to protect *data bits* against single-bit or multibit errors. Data bits are literally from the program data and *code bits* are newly introduced to form a code word with data bits. For example, a simple tuple code word can be formed as <data bit, code bit> and the *encoding scheme* can be set as *code bit = data bit*. In line with the encoding scheme, the fault-free code is then supposed to be either 00 or 11 because the code bit must be equal to the data bit. When the data bit is needed by the program, the whole code word will be read out and checked for the correctness. A fault-free code word means no error in the data bit. Otherwise, the value 01 or 10 means that a bit flip happens to the data bit or code bit due to an alpha particle or a neutron strike.

The number of bit errors a code word can detect or correct is determined by its minimum Hamming distance. The Hamming distance [73] between two words or bit vectors is defined as the number of bit positions they differ in. Given a code word space, the minimum Hamming distance of the code word is the minimum distance between any two fault-free code words. For the aforementioned example, fault-free code words are 00 or 11, thus the minimum Hamming distance for this code space is 2. Table 2.1 presents the number of bit errors that can be detected or corrected by a code word given its minimum Hamming distance. According to this table, the above discussed tuple code word can detect a single-bit error but cannot correct it. For example, it can be easily detected that a code word 01 is erroneous as it is opposite to the encoding scheme. The error correction, however, cannot be performed because it is difficult to determine whether the code word changes from 00 to 01 (due to a code bit flip) or 11 to 01 (due to a data bit flip). Such a coding scheme is defined as Single Error Detection (SED). We can also observe from the table that a coding scheme can correct single-bit errors and detect double-bit errors if its minimum Hamming distance is at least 4 – such a coding scheme is referred

to as SECDED code. Over the years, a wide range of Error Correction Code (ECC) techniques have been developed and implemented across different layers of the entire computer system. SECDED [76] is a fundamental fault-tolerant technique in the realm of ECC and is widely used due to its simple implementation. For a regular DDRx-based memory, a collection of x8 DRAM chips operating in lockstep deliver a 64-bit word on the data bus, with each chip contributing an 8-bit subset. In this case, an additional 8-bit chip is needed to provide SECDED protection.

Error coding techniques like SECDED can protect the memory system against bit failures. However, protecting large-granularity failures, such as column/row/bank-failures, requires chip-level schemes. Chipkill is an IBM trademark for an advanced ECC technique which can protect against up to a full DRAM chip failure [49, 81]. One simple implementation is to distribute the bits of a Hamming code ECC word across multiple memory chips, such that the failure of any single memory chip will affect only one code bit per word. In this way, even though an entire memory chip experiences complete failure and stops functioning, data in that memory chip can be recovered using the other chips and the ECC code. Chipkill-based correction provides stronger protection for the memory system than traditional SECDED protection. Typical implementations use more advanced codes, such as a BCH code [185], which can correct multiple bits with less overhead. IBM first introduced the concept of Chipkill-correct [49] in 1997, which interleaved the ECC coding such that two consecutive data bits were encoded in two different code words. This approach is capable of protecting the memory data against complete damage of a single memory bank. AMD further optimized this technique by achieving the same level of protection with a reduced number of required memory ranks [14]. Udipi et al. [174] proposed a novel implementation of the Chipkill-level reliability technique for future energy-efficient memories.

Our first work in this thesis targets the performance and reliability of hybrid HBM-DRAM memories. For the fault-tolerant techniques, we assume SECDED coding for on-package HBM because of its lower complexity and power consumption. In line with production systems, we assume single-Chipkill ECC for off-package DRAM.

2.2.3 Metrics

MTBF, MTTF and FIT-rate, three prevailing evaluation metrics, are widely used by computer architects and manufacturers to quantify the reliability of a system or a product. FIT rate and MTTF can be used to measure the number of failures or the number of errors [126, 127, 184]. We now review them in the context of soft errors.

Mean Time Between Failures (MTBF). MTBF is the average time between system breakdowns. This statistical value represents the average amount of time between random failures over a long period of time for a given com-

ponent. It also indicates the system reliability that is calculated from known failure rates of various components in the system. Assume n components exist in the system, the system MTBF is computed from the MTBFs of the individual components:

$$MTBF_{sys} = \frac{1}{\sum_{i=1}^{n} \frac{1}{MTBF_i}} \tag{2.1}$$

Mean Time To Failure (MTTF). MTTF represents the mean time expected until the next failure of a piece of equipment. Technically, MTBF is used in reference to a repairable item while MTTF is used for non-repairable items. However, MTBF is commonly used for both repairable and non-repairable items. The system MTTF is calculated similarly as MTBF:

$$MTTF_{sys} = \frac{1}{\sum_{i=1}^{n} \frac{1}{MTTF_i}} \tag{2.2}$$

Failure In Time (FIT) rate. FIT rate is defined as the total number of errors in a billion device hours. This term is particularly used by the semiconductor industry and is also used by component manufacturers. If the evaluated components in the system are independent, the system FIT is the addition of FIT for individual components:

$$FIT_{sys} = \sum_{i=1}^{n} FIT_i \tag{2.3}$$

FIT values can also be calculated with the formulas below with the MTTF shown in the reliability data.

$$FIT = \frac{1}{MTTF} \tag{2.4}$$

FIT rate is a typical representation of the Soft Error Rate (SER); hence we adopt this metric for the soft error reliability analysis in our work.

2.2.4 Architecture Vulnerability Factor Analysis

Recall that our first work in this thesis is about how to improve the performance and reliability of a hybrid memory system, in which the reliability analysis relies on the evaluation of system vulnerability to soft errors. Architecture Vulnerability Factor (AVF) analysis is a well-established analytical reliability estimation technique, proposed to perform a fast and accurate analysis of system reliability. In this subsection, we introduce AVF-based analysis in detail, and this method will be adopted in this thesis for further proposals.

2.2.4.1 AVF-based Analysis

Architecture Vulnerability Factor (AVF) analysis was proposed in [153] to calculate the probability that a fault within a certain architecture unit leads to an observable program error. AVF is calculated using the processor state bits of Architecturally Correct Execution (ACE) as we will demonstrate later. The AVF calculation usually involves a model of a hardware structure, for example, a performance simulator, where performance counters are used to profile and track the instructions. One representative example is to use the Asim framework [54] of the IA64 architecture [99] to estimate AVF for instruction queues and execution units. Others extended existing AVF models to estimate the system-level vulnerability factor for on-chip instruction caches [183], data caches [72], L2 caches [39] and register files [124]. Those optimized analytical models mitigate the overestimation problems by studying the system-level effects of a particular behavior on a certain architecture component. For example, Haghdoost et al. [72] improve the AVF estimation accuracy by up to 40% through taking into account both the read frequency and error masking/detection of system-level vulnerability of data caches. In parallel with architectural AVF analysis, other work takes advantage of such analysis for software reliability. Rehman et al. [148] propose compiler optimization techniques to generate reliable code which minimizes the ACE latency of program variables. The work is further extended in [149] to jointly consider functional correctness and timing reliability. In addition, Weaver et al. [184] propose both software and hardware techniques to reduce the soft error rate based on fault tracking and AVF analysis.

In summary, AVF-based analytical models are widely used and play an important role in the system reliability research. In addition, AVF analysis is fast especially compared to fault injection campaigns. The collective advantages of current AVF analysis motivate us to use such a methodology to quantify the reliability of memory objects in our work.

2.2.4.2 AVF versus SER

To calculate AVF, Mukherjee et al. [153] group all bits in a hardware structure into two types: (1) those necessary for architecturally correct execution (ACE), and (2) the remaining un-ACE bits. AVF of a hardware structure is calculated as the fraction of its ACE bits to the total number of bits over a period of time.

ACE bits. A fault in the ACE bits results in an observable program error due to lack of error correction techniques, whereas a fault in un-ACE bits has no impact on correctness. For example, all bits of the branch predictor are un-ACE because a fault in the branch predictor can never lead to incorrect updates to the architectural state of a program. Furthermore, a bit can be ACE for only a fraction of the total program execution time and un-ACE for the rest of the time. For example, a physical register is written into some bits

by an instruction at the beginning of the program execution, then the register values are read in the middle of the execution and are no longer required after that. In this case, these bits are ACE until they are required, namely, this physical register contains ACE bits for only half of the total execution time. From the architectural perspective, NOP instructions, performance-enhancing instructions, predicated instructions with a false predicate and dynamically dead instructions will produce un-ACE bits.

Architecture Vulnerability Factor (AVF). As mentioned above, the AVF for a hardware structure is the fraction of the total execution time when it is in ACE state. For example, a physical register is written at the beginning of the execution, read half-way and dead thereafter. In this case, the AVF for this register equals to 0.5 or 50%. In general, the AVF of a hardware structure H over a period of N cycles can be computed as:

$$AVF_H = \frac{\sum_{i=0}^{N} ACE_i}{B_H \times N} \tag{2.5}$$

where ACE_i represents the ACE bits in structure H at cycle i and B_H is the total bit size of this structure.

Soft Error Rate (SER). SER is the probability for an uncorrectable error resulting from incorrect program execution. For a hardware structure H, the SER is the product of its FIT rate and AVF:

$$SER_H = FIT_H \times AVF_H \tag{2.6}$$

The calculation of FIT rate has been illustrated in Section 2.2.3. The FIT value in the equation presents the probability of a transient uncorrectable error, while the AVF value captures the incidence of the application reading the erroneous bit. Putting Equation 2.5 and Equation 2.6 together, we conclude that SER of a certain structure is proportional to the ACE bits if the time period, environmental factors and circuit characteristics are kept unchanged.

2.2.4.3 AVF Estimation for Memory System

AVF analysis is performed using a simulation model of a hardware structure, and the accuracy of AVF estimation depends on the level of detail incorporated in the model [24]. The more details, the longer it takes to estimate AVF but the more accurate the estimation will be. For DRAM-based memory systems, soft errors can be caused by either neutron particles from cosmic rays or alpha particles present in the packaging material of a chip. These particles manage to penetrate the die, generate a high density of holes and electrons in its substrate, and finally create an imbalance in the device's electrical potential distribution that causes stored data to be corrupted. Corrupted data will be ACE if they are loaded and processed by instructions and further affect the program execution. Under such circumstances, accurately evaluating the vulnerability of a memory

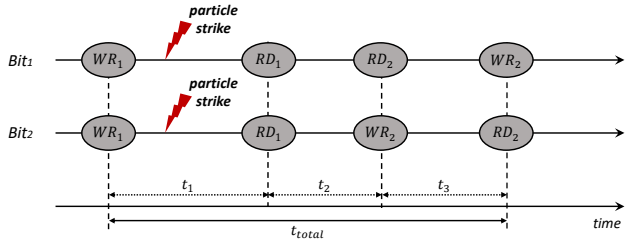


Figure 2.3: AVF of two bits in the memory system. *Two bits in the memory could have the same hotness but different AVFs depending on the sequence of reads and writes.*

system to soft errors requires monitoring every bit in the memory during the whole program execution.

The AVF of a bit is the fraction of time that the bit is in ACE state (see Equation 2.5). We illustrate an example in Figure 2.3 to show the AVF calculation for two bits in memory. Assume that both bits are read and written twice during a program execution. Bit_1 is written first, followed by two reads and finally written again. Bit_2, however, is written and read in an alternating way. Any particle strike (e.g., neutron or alpha strike) could result in incorrect execution in case of no correction before loading data. Therefore, Bit_1 is in ACE state from request WR_1 to RD_2, leading the AVF of this bit being equal to $((t_1 + t_2)/t_{total})$. Bit_2 sustains the same particle strike but has a different sequence of reads and writes. Time between WR_1 and RD_1 contains an ACE bit as the data for the first read operation RD_1 is corrupted by the particle strike. The data for the second read RD_2, however, has been masked as a correct value because it is overwritten by WR_2. Hence, the AVF for Bit_2 is equal to (t_1/t_{total}). This example demonstrates that two bits with the same number of reads and writes could potentially have very different AVF values; thus a precise calculation of AVF requires the record of each read and write operation, both the frequencies and time sequences, for every bit in memory during the whole execution.

From a practical point of view, the granularity of the AVF analysis relies on the memory access granularity. For OS-managed memory, we can perform AVF analysis on the memory system at a cache line granularity because a cache block is the fundamental unit of data transfer in memory. For the memory system with modern managed programming languages, garbage collection organizes the virtual memory heap at an object granularity, thus the AVF analysis can be performed through tracking every read and write to an object. In addition to the estimation accuracy, the AVF analysis also needs to compromise with the analysis efficiency. In our work, we find that precisely computing the AVF of an object, namely tracking all reads and writes, incurs too much overhead to perform online profiling in the context of a managed runtime environment.

Hence, we use proxy metrics that correlate well with AVF and are easier to collect. Sections 3.3 and 3.4 explain the definition and usage of AVF proxies in detail.

2.3 System Simulation

Prototyping a designed computer system is so time-consuming and inflexible that researchers and practitioners heavily rely on architectural simulation to steer future system development. Specifically, computer architects employ simulation to predict the performance and/or other characteristics concerned with a set of workloads executing on a target system. Simulation techniques can be classified into many different categories depending on the context – functional versus timing simulation; trace-driven versus execution-driven simulation; workload scaling versus architecture scaling simulation. These techniques may model the target system at different levels of abstraction, whereas they are proposed with generally two goals – improving prediction accuracy and/or reducing simulation time. We now discuss the aforementioned simulation techniques and clarify which techniques we use in this thesis.

2.3.1 Functional versus Timing Simulation

2.3.1.1 Functional Simulation

Functional simulation models the functional characteristics of a designed system. The instructions are simulated one at a time much like an interpreter, hence they are also called ISA simulators. Functional simulation can also simulate specific components of the processor, such as the cache hierarchy or branch predictor. Some simulators, e.g., Sniper [34], offer a cache-only simulation mode. Many other prevailing simulators, such as SimpleScalar [31] and Gem5 [23], also offer functional simulation models. Pin [113], a binary instrumentation framework developed by Intel for x86-applications, is widely used to build functional simulators. This tool allows to instrument an application as it is executed and collect statistics from the dynamic instruction stream. The statistics collected can be used to build functional simulators such as CMP$im [85], a functional cache simulator for multicore processors, as well as full timing-based simulators such as Graphite [121] and Sniper [34]. In summary, functional simulation is favorable to verify the functionality of a system or its certain substructures, or provide high-level statistics such as dynamic instruction counts, cache miss rate, the number of mispredicted branches, etc.

Functional simulation for a system or a specific system component is relatively fast because it only models the functional characteristics of a structure, with no timing-related details involved. However, the lack of timing-related simulations also induces its inability to deliver performance characteristics, that

is, it cannot be used to predict performance or other timing-related statistics for the target system.

2.3.1.2 Timing Simulation

Timing simulation keeps track of both functional characteristics and timing-related details of a system. It simulates the system and evaluates applications in an cycle-accurate way, further categorized into cycle-accurate simulation and cycle-level simulation.

Cycle-accurate simulation models every component of the target system in software, that is, simulating the instruction set, the pipeline and the memory hierarchy of a system on a cycle-by-cycle basis. This detailed architectural simulation generates accurate performance predictions compared to the real system. However, such an accurate simulation is also accompanied by two disadvantages. The first disadvantage is the overwhelming overhead of simulation time. For example, simulating an instruction may take up to several milliseconds in the simulator while it only takes one cycle in real hardware, leading to a slowdown of several orders of magnitude. In addition, it is very difficult for the academia to do cycle-accurate simulations of commercial products. The processor company would keep simulators inside rather than release them, to prevent third-parties from figuring out the design details of their products. This action is intelligible in consideration of their research and development costs, and profit needs. In conclusion, it is infeasible for computer architects (especially in academia) to explore processor designs using cycle-accurate simulator either for its unavailability or unendurable simulation time overhead.

Fortunately, some alternative simulators have been proposed to academics to perform cycle-level simulations. These simulators will provide high detail for certain components of the system and do a fast and less accurate simulation of the rest of the architecture. For example, Gem5 [23] simulates a target system based on events rather than cycles, that is, the simulation directly jumps to the time when an event is scheduled instead of going through all cycles. In this way, it reduces the simulation time by not simulating cycles with no scheduled events. On the other hand, Gem5 tracks events on a cycle-to-cycle basis and its 'Minor' (in-order) and 'O3' (out-of-order) CPU models allow for simulating the pipeline in detail, both keeping its accuracy at a high level. Interval simulation [65] is proposed to simulate multicore systems at a higher level of abstraction compared to detailed cycle-level simulation. Carlson et al. [34] further integrate the interval simulation methodology into Graphite [121], a parallel simulation infrastructure, to build a fast and accurate multicore simulator called Sniper [34]. ZSim [156] also implements high-abstraction simulation models and it focuses more on simulating memory hierarchies and many core heterogeneous (single-ISA) systems. Other example simulators include SimpleScalar [31], PTLSim [192], etc. Overall, the cycle-level simulation achieves a significant simulation speedup compared to the cycle-accurate simulations while still maintaining a high level of prediction accuracy.

Timing simulations offer different trade-offs in terms of prediction accuracy and simulation time. However, their prominent simulation time still makes them inappropriate to predict performance for emerging applications, especially high-level programming applications executing on a large-scale target system.

2.3.2 Trace-Driven versus Execution-Driven Simulation

Architectural simulation can be categorized into functional simulation and timing simulation according to the level of simulation details just described in Section 2.3.1. It can also be classified according to the types of input. More specifically, the input can be a trace collected from an execution of a program on a real microprocessor (so called trace-driven simulation) or a program itself (so called execution-driven simulation).

2.3.2.1 Trace-Driven Simulation

Trace-driven simulation [175] loads a fixed sequence of trace records from a file as an input to a simulator. The information in the trace file depends on the simulation target. For instance, simulating the cache hierarchy only needs to record information of memory accesses. When it comes to a detailed system simulation, the trace files need to contain information for the entire system, such as memory references, branch outcomes, dynamically executed instructions, among others.

The trace files fed into the simulator are first generated using a tracer or profiler, which can be done offline or along with the simulation. Most of these files only record instructions that were successfully completed and lack instructions that were executed on the wrong path after a branch misprediction. Such a deterministic simulation makes trace-driven simulation comparatively fast, and its results are highly reproducible. However, it is not always representative of the real execution of an application for lack of information on the wrong path. On the other hand, detailed trace recording for an accurate simulation requires a very large storage space, especially for long-running applications [66].

2.3.2.2 Execution-Driven Simulation

Execution-driven simulation [31] directly reads instructions for the evaluated program as input and simulates the execution of machine instructions on the fly. Since the input file contains instructions that are not necessarily executed, it is easier to simulate wrong-path instructions in case of a branch misprediction in contrast to trace-driven simulation. In addition, the input file for execution-driven simulation only contains static instructions for a program and every instruction appears only once in the file, which makes it typically several magnitudes smaller than a trace file. The benefits of storage space

obtained from the input program file come up with an overhead for the simulation time. Specifically, the execution-driven simulation is much slower than the trace-driven simulation because it has to process each instruction one-by-one and update the status of all microarchitecture components involved.

In summary, the selection of input types for simulation is a trade-off between storage space and simulation time. A very detailed trace for a highly accurate simulation requires a very large storage space, whereas a very accurate execution-driven simulation takes a very long time to execute all instructions in the program. Computer architects can propose or select different simulators for their specific research purposes. Shade [46], MASE [103], Synchrotrace [157] and TaskSim [150] are representative examples of the trace-driven simulators. Typical execution-driven simulators include SimpleScalar [18], SPIM [138], PTLSim [192], ESESC [17] and Fast [40].

2.3.3 Workload versus Architecture Scaling Simulation

Speeding up simulation can often be achieved through scaling. Existing approaches have mainly focused on scaling the workload to be simulated or raising the abstraction level of the simulation models. Another promising dimension is to reduce the simulation scale from an architecture perspective. We now discuss these scaling techniques in detail.

2.3.3.1 Workload Scaling

Scaling down the workload opens up a new avenue to speed up simulation from the perspective of workload abstraction. Sampling is widely used to select representative regions of an application to execute and extrapolate the simulation results to the complete application execution. There are a wide range of sampling approaches previously proposed to explore the application abstraction space.

Random sampling, firstly proposed by Conte et al. [47], selects an unbiased set of regions randomly and simulates them in a detailed way. Systematic sampling, on the other hand, can produce unrepresentative samples if a program exhibits periodic behavior. A well-known example of systematic sampling is SMARTS [188] which employs periodic statistical sampling. Both methods assume that they are statistically representative with enough samples selected throughout the entire application execution. The effectiveness of this assumption is supported by the central limit theorem (CLT) [108] which implies that the sampled mean of performance-related metrics will approach the overall mean with a sufficiently large number of samples.

Other sampling approaches exhibit more characteristics of the target application. A straightforward approach is to select samples based on certain microarchitectures, such as functional cache and branch simulations [167].

Its limitation is that such sampling depends on the observed microarchitectures, leading the selected samples possibly being unrepresentative for other microarchitectures. SimPoint [161] overcomes this limitation by selecting microarchitecture-independent metrics and detecting phase behavior in applications. To be specific, SimPoint divides the execution of an application into fixed intervals; it assumes that intervals with similar basic block behavior, called a *phase*, will exhibit similar microarchitecture behavior and thus only one interval per phase needs to be simulated. Phase detection and classification can be done through either offline profiling based on basic block vectors [142, 161] or dynamic branch profiling [162]. The SimPoint approach is further employed by Patil et al. [137] to enable deterministic sample replay using Pin [113].

2.3.3.2 Architecture Scaling

Architecture scaling can be achieved by either abstracting away certain details of the processor microarchitecture or constructing down-scaled models to reduce the simulation scope.

Several models have accelerated the simulation through raising the level of abstraction. For example, interval simulation [65] models the impact of miss events on performance through mechanistic analytical modeling instead of detailed simulation. ZSim [156] and Sniper [34] construct high-abstraction simulation models for superscalar processors. MUSA [68] combines multiple levels of simulation detail, ranging from cycle-accurate micro-architectural simulations to high-level analytical models. The One-IPC model assumes that a single instruction is executed per cycle in the absence of miss events such as cache misses and branch mispredictions, which is further employed in CMP$im [85] and Graphite [121]. BookSim [90] is very fast using cycle-accurate simulation whereas it only models the network components.

Down-scaled simulation aims to construct a down-scaled model of the target system while maintaining its key characteristics. It is difficult to build an exact miniature of a target system, especially in the field of modern processor architectures. Eyerman et al. [59] proposed a down-scaled model for an experimental Intel processor, called the Programmable Integrated Unified Memory Architecture (PIUMA), which is specifically designed for the efficient execution of graph analytical workloads. They simulate each component of the target large-scale system in detail but on a smaller scale to limit the simulation overhead. This work provides preliminary research on constructing scale models for modern processors, even though it is limited to be widely used for domain-specific designs since it lacks a method for modeling shared resource contention as we generally observe in general-purpose multi-core processors.

2.4 Performance Modeling

An alternative approach to simulation is to use mathematical models to capture performance-related characteristics and component interactions in a processor. The performance modeling itself does not simulate any component of the processor, although it often requires simulation results as input to train the prediction model. Performance modeling can be generally classified into two types, namely machine learning based modeling and analytical modeling.

2.4.1 Machine Learning based Modeling

Machine Learning (ML) based modeling constructs prediction models using training data obtained from detailed architecture simulation and then evaluates metrics for the target system through prediction models. This approach is established based on the premise that machine learning can predict the behavior of modern processors using sufficient input information and effective training techniques. ML-based modeling avoids simulating complex (micro-)architectures of target processors. Two steps are required for ML-based modeling – prediction model construction and target system evaluation.

It is relatively easy to design a suitable machine learning approach to build a prediction model. Another thing is to collect sufficient information to train the prediction model. The collection of training information, however, is time-consuming and difficult to operate because: (1) we need massive training examples obtained from detailed simulations; and (2) these training inputs should reflect the impact of system components and involve characteristics of applications running on the processor. Once the prediction model is built, it is fast to evaluate target systems using the prediction model.

A large number of ML-based performance modeling techniques have been proposed over the years. For example, Lee et al. [104] and Ipek et al. [82] build accurate performance prediction models using regression models and artificial neural networks, respectively. These models are further extended to a large design space. Lee et al. improve their previously proposed models, and apply the optimized models to design space evaluation [105] and performance modeling for parallel applications [106]. Azizi et al. [20] include power and energy metrics to find power-performance trade-offs. Singh et al. [166] explore the possibility of using performance counters on real hardware to perform real-time power modeling and thread scheduling. However, predicting performance for larger-scale modern systems is still beyond the reach of these approaches.

2.4.2 Analytical Modeling

Analytical modeling portrays application behavior on an evaluated design through mathematical equations and algorithms like probabilistic methods, queuing theory, etc. The foundations of these modes are the fundamental

understanding of the evaluated systems. To be specific, they are constructed through observing how applications behave on a designed platform and then modeling these interactions using evaluation metrics such as instruction dependency, cache miss rate, branch misprediction rate, etc. Evaluation metrics can be derived from a functional simulation and served as inputs to the mathematical prediction model.

Over the years, a significant amount of research has been performed to model processor performance using different analytical models. Emma et al. [55] involve CPI stacks in the analysis of performance bottlenecks. Michaud et al. [118] quantify the influence of instruction fetch bandwidth on performance with respect to branch mispredictions and instruction-level parallelism (ILP). Hartstein et al. [74] propose a model that details how the optimal pipeline length can change as function of the ILP and pipeline stalls. A first-order model focusing on pipeline stalls due to miss events is developed by Karkhanis et al. [97]. The interval model is applied to the general problem of resource scaling in out-of-order superscalar processors [58].

Analytical models are much faster and generally less accurate than ML-based models as no more detailed simulation is required for the whole system. The efficiency of analytical modeling makes it an attractive approach to quickly explore large design spaces at their early stages in the design cycle. However, an intrinsic downside of such a technique is its limited capability on deep exploration and thorough analysis for a specific proposed architecture – for example, how to analytically model overlap effects as well as timing-sensitive events in large target systems.

2.5 Java Workload Benchmarking

Managed programming languages such as Java, C#, JavaScript and Python have been increasingly popular among programmers. These languages provide a range of services, such as garbage collection, zero initialization and just-in-time (JIT) compilation, to the programmer. Such services are involved in a managed runtime environment. In Chapter 3 and Chapter 5, we consider nine Java applications from the DaCapo suite [29] to evaluate our proposals and use Jikes Research VM (RVM) 3.1.2 [10, 11] as the platform.

Jikes RVM is a Java-in-Java VM that involves a baseline compiler and a JIT optimizing compiler. Non-determinism is introduced into Java performance evaluation due to the timer-based sampling for JIT optimization. To evaluate Java workloads rigorously, we use replay compilation [79, 155] to eliminate the non-determinism introduced by the JIT compiler. More specifically, replay compilation requires a *profiler* and a *replayer*. The profiler records the profiling information used to drive the compilation decisions and a single *compilation plan* is determined from these decisions. The compilation plan then forces the VM to compile each method to a predetermined optimization level (for

the run with lowest execution time) in the replay run. During the replay phase, a benchmark is iterated twice within a single VM invocation. The first iteration applies the optimization plan to each method, which incurs a compilation overhead. The second iteration does not recompile methods, which excludes the compilation overhead and represents the steady-state behavior of the application. We take our measurements during the second iteration. For all experimental results of Java workloads in this thesis, we run each application four times and report the arithmetic mean in the figures.

Chapter 3

Reliability-Aware Garbage Collection for Hybrid HBM-DRAM Memories

Emerging workloads in cloud and data center infrastructures demand high main memory bandwidth and capacity. Unfortunately, DRAM alone is unable to satisfy contemporary main memory demands. High-bandwidth memory (HBM) uses 3D die-stacking to deliver 4–8× higher bandwidth. However, HBM has two drawbacks: (1) capacity is low, and (2) soft error rate is high. Hybrid memory combines DRAM and HBM to promise low fault rates, high bandwidth, and high capacity. Prior OS approaches manage HBM by mapping pages to HBM versus DRAM based on hotness (access frequency) and risk (susceptibility to soft errors). Unfortunately, these approaches operate at a coarse-grained page granularity, and frequent page migrations hurt performance.

This chapter proposes a new class of reliability-aware garbage collectors for hybrid HBM-DRAM systems which place hot and low-risk objects in HBM and the rest in DRAM. Our analysis of 9 real-world Java workloads shows that: (1) newly-allocated objects in the nursery are frequently written, making them both hot and low-risk, (2) a small fraction of the mature objects are hot and low-risk, and (3) allocation site is a good predictor for hotness and risk. We propose RiskRelief, a novel reliability-aware garbage collector that uses allocation site prediction to place hot and low-risk objects in HBM. Allocation sites are profiled offline and RiskRelief uses heuristics to classify allocation sites as DRAM and HBM. The proposed heuristics expose Pareto-optimal trade-offs between soft error rate (SER) and execution time. Compared to a state-of-the-art OS approach for reliability-aware data placement, RiskRelief eliminates all page migration overheads, which substantially improves performance while

delivering similar SER. Reliability-aware garbage collection opens up a new opportunity to manage emerging HBM-DRAM memories at fine granularity while requiring no extra hardware support and leaving the programming model unchanged.

Section 3.1 elaborates on the dilemma of current memory systems, the previous improvements and limitations, and our key contributions to this field. Section 3.2 characterizes the performance and reliability for HBM and presents currently widely-used approaches to manage HBM, which is followed by the background in soft error reliability and managed runtimes in Section 3.3. In Section 3.4, we explore the characteristics of object hotness and risk in Java applications and find that allocation site is a good predictor for object hotness and risk. We then propose reliability-aware garbage collection (called RiskRelief) for hybrid HBM-DRAM memory systems and describe RiskRelief in detail in Section 3.5. Section 3.6 introduces the simulation and emulation methodology used in the following experimental sections. The performance and reliability experimental results are presented in Section 3.7 using a detailed architectural simulation, and we report similar metrics with emulation results on real hardware in Section 3.8. Section 3.9 complements some other related work targeting hybrid DRAM-PCM memory systems, followed by the conclusion in Section 3.10.

3.1 Introduction

Emerging cloud workloads, such as machine learning inference and stream analytics, have encouraged new throughput-oriented compute platforms. These platforms consist of many-core processors, graphic processing units, and a range of accelerators. Altogether, these compute platforms have an insatiable demand for main memory bandwidth. The confluence of ever-growing compute power and the slow historical growth in pin count for off-chip communication [83] has exacerbated the memory bandwidth wall [152]. High Bandwidth Memory (HBM) [13], i.e., 3D-stacked DRAM, delivers higher bandwidth than traditional DRAM, while consuming less power and space [25, 43, 44, 51, 70, 88, 91, 164].

HBM has two shortcomings though: (1) capacity is limited to a couple GBs, and (2) soft error rate is high due to higher density and new failure modes [84, 130]. Hybrid HBM-DRAM memory combines the best of both worlds to provide high capacity and high bandwidth. Unfortunately, unless properly managed, HBM reliability is a concern. Our experimental results reveal that an HBM-Only system yields 34% higher performance than a DRAM-Only system, but the entire program heap is capacity-limited and, moreover, is highly vulnerable to soft errors. A DRAM-Only system, on the other hand, is substantially more reliable (by at least two orders of magnitude), but at the expense of considerably lower performance compared to HBM-Only. The goal of this work is to achieve the best of both worlds, i.e., deliver high reliability while achieving high performance.

A flurry of prior work proposes hardware and OS approaches to optimize hybrid memory performance. Specifically, hardware approaches use HBM as a cache for DRAM [43, 44, 89, 96, 107], whereas OS approaches map frequently accessed pages in HBM [135, 145, 146, 165]. Only recently have researchers turned attention to data placement approaches to address the low reliability of HBM [69]. Indeed, soft error rates in production systems are continuously increasing, and they grow proportionally with information density [98]. Hardware-only approaches to tackle reliability are insufficient because they will soon require impractical error detection and correction capabilities [117]. OS approaches [69] also face drawbacks: (1) they are reactive, (2) page migrations incur significant performance penalty, and (3) they are coarse-grained and require excessive HBM capacity.

This chapter takes a different, so far unexplored, approach by leveraging garbage collection in modern managed languages to place program data in hybrid HBM-DRAM memory at a finer granularity than state-of-the-art OS approaches. Garbage collection (GC) in managed languages such as Java, C#, JavaScript, Python, and Ruby manages virtual heap memory on behalf of the programmer. Most high-performance GCs place newly allocated (young) objects in a small nursery space. A nursery collection copies surviving objects to the mature space. This generational heap organization leads to short pause times and high application (mutator) locality and performance [15]. Our analysis of various Java applications from the DaCapo suite [29] shows that: (1) nursery objects are hot (frequently accessed) and low-risk (highly mutated), and (2) only a small fraction of nursery survivors are hot and low-risk. These results reveal an opportunity to effectively manage HBM-DRAM memory.

This work proposes a new class of *reliability-aware garbage collectors* for hybrid memory. These collectors place hot and low-risk objects in HBM to improve reliability and performance. The remaining objects are placed in DRAM to utilize its large capacity. Reliability-aware garbage collection overcomes the disadvantages of the state-of-the-art OS approach. Specifically, prediction enables pro-active allocation of objects in HBM as opposed to reactive page migrations. Moreover, placing objects using GC eliminates the overhead of costly page migrations.

In this chapter, we propose two reliability-aware garbage collectors. RiskRelief-Nursery (RR-N) places the nursery in HBM and the mature space in DRAM. It requires minimal changes to the Java runtime but is highly effective in delivering low soft error rates compared to an HBM-Only system, while improving performance compared to a DRAM-Only system. RiskRelief-Mature (RR-M) places the nursery in HBM and exploits offline program profiling to place hot and low-risk nursery survivors in HBM. We show that mature object hotness and risk are predictable on a per allocation-site basis. Surprisingly perhaps, we find that object hotness and risk are weakly correlated. Hence, placing objects in HBM based solely on hotness significantly hurts reliability. The insight is to place objects in HBM versus DRAM based on hotness *and* risk.

Based on these observations, we propose heuristics to classify allocation sites as DRAM and HBM. Allocation sites are classified as HBM if most objects they allocate are hot and low-risk. All other allocation sites default as DRAM. We generate this per allocation-site advice offline and feed it to RR-M. In turn, RR-M uses the advice during runtime to place nursery survivors in HBM or DRAM. Our proposed heuristics expose previously unseen Pareto-optimal trade-offs between execution time and soft error rate. A single profiling run generates a range of advices for the GC runtime. Thus, depending upon factors such as environmental conditions, available HBM capacity and performance goals, a system operator can adjust the advice fed to RR-M to meet specific constraints.

Our experimental results show that RR-N reduces the overall soft error rate by 18× on average compared to an HBM-Only system, while improving performance over a homogeneous DRAM-Only system by 20%. The state-of-the-art OS solution by Gupta et al. [69] achieves similar SER as RR-N, however, performance is substantially worse (even worse than the DRAM-Only system) because of the high cost of TLB shootdowns on modern x86 multicores [135]. Both RR-N and the prior OS approach use a modest 128 MB of HBM on a 32-core platform. RR-M uses an additional 18% of HBM capacity but delivers 29% higher performance compared to a DRAM-Only system. Higher HBM capacity impacts overall SER and RR-M reduces SER by 9× over HBM-Only.

In summary, the main contributions of this chapter are:

- hotness (access frequency) and risk (susceptibility to soft errors) characterization of objects in Java applications, showing that hotness and risk are only weakly correlated;
- showing that allocation site is a good predictor for object hotness and risk;
- the design and implementation of reliability-aware garbage collection for hybrid HBM-DRAM memories to minimize soft error rate while maximizing overall application performance — in contrast, performance-optimized HBM-DRAM management significantly hurts reliability;
- profile-driven RiskRelief reliability-aware collectors that exploit allocation-site prediction to place hot and low-risk objects in HBM and the rest in DRAM;
- a profiling framework to measure object hotness and risk on a per allocation-site basis; two heuristics to generate the allocation advice for GC; and a compilation framework that exploits the advice to steer allocation of objects in HBM and DRAM.
- simulation and real hardware emulation results motivating hybrid HBM-DRAM memory for Java applications, and showing that RiskRelief collectors manage hybrid HBM-DRAM memory significantly better than state-of-the-art OS approaches.

3.2 Exploiting High-Bandwidth Memory

In this section, we discuss the motivation for HBM, and we describe its distinct performance and reliability characteristics. We also review existing approaches to manage HBM.

3.2.1 3D-Stacked Memory

Disruptive approaches to mitigate the memory bandwidth wall are needed [152]. The bandwidth between conventional DRAM and the processor is limited by pin count, which increases by roughly 10% every year [83]. However, compute power grows much more rapidly. Furthermore, having enough pins to stream a 1024-bit word every cycle to the processor would require 40 Watt just for memory I/O [119]. High-bandwidth memory vertically stacks DRAM chips in a 3D arrangement to deliver higher bandwidth than conventional DRAM. Through-silicon vias (TSVs) interconnect the vertically stacked chips using wide communication lanes.

Conventional DRAM technology, e.g., DDR4, places two 64-bit words on the data bus every cycle. Several DRAM chips work in tandem to produce the word. For example, 16 $\times$4 chips each provide 4 bits every cycle to render a 64-bit word. In contrast, the state-of-the-art HBM standard allows up to 12 dies per stack, and each stack has 8 unique 128-bit channels per stack, leading to a much wider, 1024-bit memory interface [86]. Internally, each DRAM chip consists of many banks. A 64-byte cache line is striped across banks in different DRAM chips to maximize parallelism. Hardware employs error correction codes (ECC) to shield against soft errors. Typically, an additional chip provides ECC protection to the data word. Most commonly, DRAM employs single-error correcting, double-error detecting (SECDED) codes.

HBM inherits the failure modes of conventional DRAM because it uses a similar cell technology and array layout. Unfortunately, new failure modes exist in HBM, for instance, due to TSV failures [84]. HBM also exhibits higher bit density increasing susceptibility to soft errors [13, 69, 84, 87]. Furthermore, HBM employs weaker error correction due to cost and complexity constraints [69, 87]. Put together, HBM reliability is a major concern which necessitates hardware and software approaches to mitigate the vulnerability to soft errors in HBM and improve the overall reliability of the memory system.

3.2.2 Managing HBM in Hardware

Exploiting HBM as a last-level DRAM cache is predominant. In particular, prior work proposes new organizations for DRAM caches [88], intelligent tag placement (for example, co-locating tags with data) [70, 96], new techniques to reduce the bandwidth consumed by cache operations [43, 44], and techniques to enable set associativity in giga-scale DRAM caches [191]. Prior work also

attempts to mitigate the performance overhead of DRAM caches for capacity-limited applications [42]. Although transparent to the software stack, DRAM caches have two drawbacks: (1) they limit the available memory capacity, and (2) they require extensive hardware support because conventional SRAM-based cache organizations are suboptimal for DRAM technology. Moreover, none of this prior work considers the low reliability of HBM, thus rendering program data in HBM highly vulnerable to transient faults. Liu et al. [112] propose Binary Star which coordinates the reliability schemes in the 3D DRAM LLC versus main memory to improve the reliability of the overall memory hierarchy. Binary Star achieves high reliability for the overall memory system with limited performance loss, while requiring modifications to both system software and hardware. RiskRelief does not require any hardware changes.

ECC codes are the first line of defense against transient faults. DRAM scaling relies on ECC hardware because smaller DRAM cells are more susceptible to soft errors. Several works study DRAM soft error rates in the field [159, 170, 171]. Weaker ECC is typically deployed in die-stacked memory due to implementation costs [87] and thus requires soft error mitigation from other sources, e.g., through software, as we discuss next.

3.2.3 Managing HBM in the OS

Existing OS approaches to manage hybrid HBM-DRAM memory aim at either maximizing performance or balancing performance and reliability. Performance-focused approaches hurt reliability [146], because they place all hot pages in HBM while being agnostic to soft error vulnerability. Gupta et al. [69] propose a dynamic page migration scheme which estimates page hotness and risk using performance counters and which migrates (every 100 ms) cold and high-risk pages to DRAM, and hot and low-risk pages to HBM. In contrast, we estimate hotness and risk at a much finer granularity of objects. Our solution pro-actively places objects in HBM versus DRAM, and does not require dynamic monitoring nor additional performance counter hardware. We compare to the OS page migration approach in this work.

Oskin and Loh [135] propose OS-managed DRAM caches. Their work shows the high cost of page migrations due to TLB shootdowns. They also explore statically partitioning program data in C applications in DRAM and HBM, albeit with negligible benefits. Their proposal does not consider the heterogeneity in reliability in a hybrid HBM-DRAM memory system. We expose both DRAM and HBM to the OS to exploit full memory capacity. Furthermore, this is the first work to expose 3D-stacked memory to garbage collection in the managed runtime for fine-grained object placement.

3.3 Background

Before describing how RiskRelief predicts hotness and risk and leverages these predictions to manage hybrid HBM-DRAM systems, we first provide additional background in soft error reliability and managed runtimes.

3.3.1 Soft Error Reliability

RiskRelief builds upon two notions, namely hotness and risk. Intuitively, hotness refers to how frequently an object is accessed, whereas risk refers to how susceptible an object is to soft errors. We now define both concepts and focus on risk more because it is a less well-known metric.

Hotness. Hotness is a well-known concept and typically refers to how frequently a particular code segment executes. Analogously, we define the hotness of an object as to how frequently the object is accessed through read or write operations. We define an object's hotness as the sum of reads and writes to the object. Our analysis shows that of all accesses to objects, 54% of the accesses on average are reads, and 46% are writes. The high percentage of writes motivates our hotness criteria as the sum of reads and writes.

Risk. Quantifying the risk of an object in HBM is more involved. We build upon the mechanistic notion of architectural vulnerability factor (AVF) to quantify susceptibility to soft errors. AVF is the probability that a transient fault leads to an observable program error. To compute AVF, Mukherjee et al. [153] categorize all bits in a hardware structure into two types: (1) those necessary for architecturally correct execution (ACE), and (2) the remaining un-ACE bits. A fault in the ACE bits results in an observable program error (assuming the fault evades ECC hardware), and a fault in un-ACE bits has no bearing on program correctness. A bit can be ACE for only a fraction of the total execution time. The AVF of a hardware structure is the fraction of all bits that are in ACE state during each cycle.

Precisely computing AVF of an object requires tracking every read and write operation. Consider an object O, stored at memory location M, is written at time t_1 and read at times t_2 and t_3, after which O is dead from the program's point of view (i.e., no other memory location points to O). O is ACE for $t_3 - t_1$ time units, namely between the write at t_1 and its last read at t_3. In case the object would have been written at times t_2 and t_3, the object would be un-ACE throughout. In other words, to precisely compute the AVF of an object, one needs to track all reads and all writes, which is too high overhead to do online in the context of a managed runtime.

Instead, we build upon prior work [69] and use proxy metrics that are easier to collect while correlating well with AVF. The proxies considered are the writes to reads ratio (W_r/R_d) and the writes-squared to reads ratio (${W_r}^2/R_d$). The intuition behind these proxies is that an object that is written a lot is more

likely to lead to more un-ACE periods. We use the writes-squared to reads ratio in this work because it places extra emphasis on the absolute number of accesses [69]. Since writes-squared to reads ratio is inversely proportional to AVF, we refer to it as AVF-X. In other words, a high writes-squared to reads ratio (high AVF-X) means low risk, and vice versa. Soft error rate (SER) is defined as the product of a device's failure-in-time (FIT-Rate) and AVF. FIT-Rate is defined as the raw failure rate due to single event faults, and depends on environmental factors and circuit characteristics.

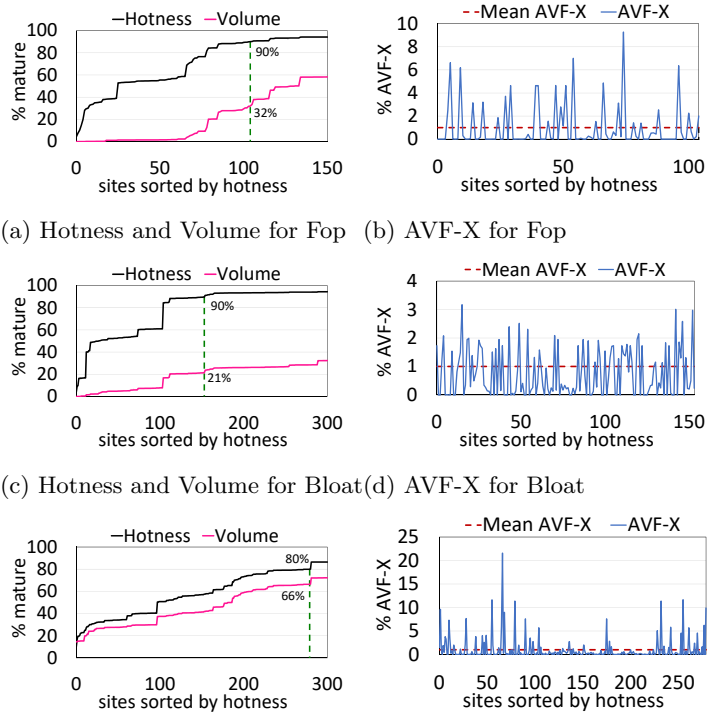
3.3.2 Managed Runtimes

Java Virtual Machine. This work uses the language runtime to improve system reliability in hybrid memory systems. Our work generalizes to languages with garbage collection, but we use the Java Virtual Machine (JVM) in this work. Exposing HBM to the JVM entails extending the OS NUMA interface [6]. We use the open-source Jikes Research VM (RVM) as our platform. Jikes RVM's modular design makes it easy to modify [10, 11, 27, 62]. Jikes RVM is a meta-circular VM written in Java. It has both a baseline and an optimizing compiler, along with several garbage collectors [26, 28, 160]. The object layout and metadata, and a variety of reference barriers can be changed quickly because of the clean interface between the compiler and garbage collector [62, 190].

Generational Garbage Collection. Despite other differences, garbage collectors in modern languages have converged on a generational heap organization. The generational organization delivers high performance because many objects die young [176]. The application (*mutator*) allocates new objects contiguously into a nursery. When the nursery memory is full, 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. The nursery space is claimed en masse for fresh allocation.

Nursery size. Nursery size is critical to overall performance, pause time, and space efficiency [15, 28, 177, 194]. A nursery collection incurs a fixed cost to scan the root set and a variable cost depending upon the number of objects that survive a minor collection. Large nurseries sometimes improve performance because objects have more time to die. They, however, increase the overall memory footprint, often unnecessarily retaining dead short-lived objects, and they incur high pause times [132, 194]. We use a 4 MB nursery because prior work establishes that it performs well for our applications [29, 158].

GenImmix. We build on the best-performing collector in Jikes RVM: generational Immix (GenImmix) [26]. We use it as the baseline and modify it to create the RiskRelief collectors. GenImmix uses a copying nursery and a *mark-region* mature space. The mark-region mature space consists of a hierarchy of blocks



(a) Hotness and Volume for Fop (b) AVF-X for Fop

(c) Hotness and Volume for Bloat (d) AVF-X for Bloat

(e) Hotness and Volume for Pmd (f) AVF-X for Pmd

Figure 3.1: Distribution of hotness and mature heap volume by allocation site (left column), versus risk for the top hottest allocation sites (right column) for Fop (top), Bloat (middle), and Pmd (bottom).

and lines. Blocks are multiples of page sizes and constitute multiple lines. Lines are multiples of cache line sizes. Objects can span lines but not blocks. Nursery collections copy nursery objects consecutively in space into free lines within blocks in the mature space by incrementing a bump pointer equal to the size of the object. This contiguous allocation outperform free-list allocators due to its locality benefits [26, 28, 79]. Immix reclaims memory at a line and block granularity by marking lines and blocks live when it marks objects live during tracing. To defragment blocks, it combines marking with copying based on runtime heuristics. We use the default settings for the maximum object size (8 KB), for line size (256 bytes), and block size (32 KB). The JVM manages objects larger than an 8 KB threshold separately, allocating them directly into a non-copying large object space [94].

3.4 Hotness and Risk Prediction

This section motivates allocation-site prediction for object hotness and risk.

3.4.1 Distribution of Hotness and Risk

We start by quantifying hotness and risk across allocation sites for three benchmarks that are representative for the entire benchmark suite, namely Fop, Bloat and Pmd. Figure 3.1 (left column) shows the cumulative distribution of mature-object hotness and their total volume (as a percentage of total mature allocation) per allocation site. Allocation sites are sorted on the horizontal axis by their hotness. We observe that a large fraction of mature-object accesses are captured by a relatively small fraction of the mature heap. For example, for Fop, 90% of the mature-object accesses are concentrated to only 32% of the mature heap. This result suggests an opportunity to allocate the relatively small fraction of hot objects in HBM to improve performance while placing the bulk of the mature heap in DRAM to exploit its capacity. Unfortunately, using hotness as the sole criterion to place objects in HBM versus DRAM severely compromises a program's vulnerability to soft errors. The graphs in the right column of Figure 3.1 report AVF-X for the objects allocated from the top-100 hot allocation sites. To provide a point of reference, we also report mean AVF-X across all mature objects. We observe a remarkable variation in AVF-X across allocation sites from well below to well above the mean. It is clear from these graphs that hotness does not imply low risk, i.e., a hot object may be high-risk or low-risk. In other words, hotness is not predictive for risk. This result implies that using hotness alone as a criterion to classify allocation sites as low- versus high-risk severely compromises soft error vulnerability. Instead, we need a method that classifies allocation sites for both hotness and risk combined, which is what we describe next.

3.4.2 Allocation-Site Homogeneity

The key insight that underpins RiskRelief is that allocation site is a good predictor for both hotness *and* risk. To demonstrate this is indeed the case, we first compute the hotness and risk for all objects and we determine which objects are among the top 10% (cutoff-threshold) for either criterion. More specifically, we label an object as hot if it is among the 10% hottest objects; if not, the object is classified as cold. Similarly for risk, we label an object as low-risk if it is among the 10% lowest-risk objects; otherwise, the object is classified as high-risk. We then compute for each allocation site, the fraction hot versus cold objects, the fraction low-risk versus high-risk objects, and the fraction of objects that are both hot and low-risk (i.e., combined). We define *homogeneity* of an allocation site with respect to hotness, risk or combined hotness/risk, as the fraction of objects that are classified in the same category. For example for the combined metric, perfect (100%) homogeneity means that all objects allocated from this site are both hot and low-risk, or they are not, i.e., they are either cold or high-risk. On the other hand, a value of 50% means no homogeneity, i.e., 50% of objects are hot and low-risk, whereas the remaining 50% is either cold or high-risk.

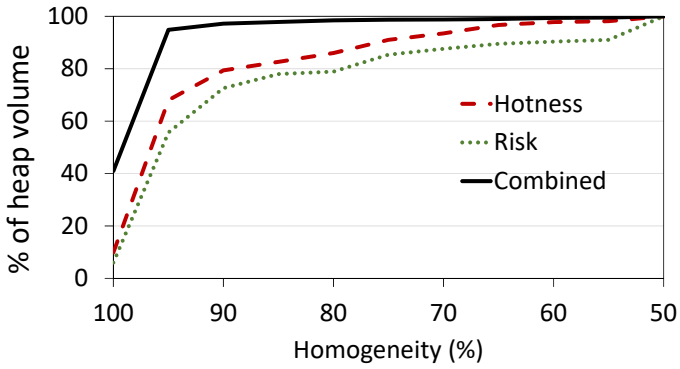


Figure 3.2: Percentage heap volume as a function of allocation-site homogeneity for hotness, risk, and combined hotness and risk assuming a 10% cutoff threshold.

Figure 3.2 reports the percentage heap volume as a function of allocation site homogeneity for hotness, risk, and the combined metric; we report average results across all benchmarks. This graph shows the fraction of heap volume allocated by sites that have a homogeneity of at least N%, with N varying from 100 to 50%. The higher the fraction heap volume covered, the better. As expected, heap volume increases with decreasing allocation site homogeneity. At 100% homogeneity, a relatively small fraction of the total heap volume is covered. However, reducing homogeneity quickly increases the heap volume covered. At 50% homogeneity, the entire heap is covered. The most important, and perhaps surprising, insight from this graph is that the combined metric outperforms the isolated hotness and risk metrics. For example, for 90% homogeneity, more than 97% of the heap is correctly classified for the combined metric, versus 79% and 72% for hotness and risk, respectively. This implies that *allocation site is a more accurate predictor for hotness and risk combined, than for hotness and risk in isolation.* The intuition is that fewer objects satisfy both the hotness and risk thresholds. We thus conclude that allocation site is a very accurate predictor to predict whether objects are hot and low-risk for placement in HBM.

Note that high allocation site homogeneity does not imply that the majority of objects are both hot and low-risk. In fact, an allocation site can have high homogeneity but produce predominantly cold objects, or produce predominantly high-risk objects, or produce predominantly hot and low-risk objects. We only want allocation sites that allocate hot and low-risk objects to be classified as HBM. We find that RiskRelief is sensitive to the object hotness and risk cutoff threshold, but is rather insensitive to the allocation site homogeneity threshold. We use a default object hotness and risk cutoff threshold of 20% and explore its sensitivity in the evaluation section. We use an aggressive allocation site homogeneity threshold of 1% to classify allocation sites as HBM

that produce even a small fraction of hot and low-risk objects, i.e., at least 1% of the objects allocated from this site are both hot and low-risk. (Note that because of high allocation site homogeneity, this implies that most objects are hot and low-risk.) We choose this aggressive threshold to make sure that hot and low-risk objects are allocated in HBM to the extent possible.

3.5 Reliability-Aware Garbage Collection

Reliability-aware garbage collection places hot and low-risk objects in HBM, and the rest in DRAM. We first provide a general overview of RiskRelief after which we describe the different components in more detail.

3.5.1 Overview

Figure 3.3 shows the workflow of RiskRelief. We first profile the Java application to collect per-object read- and write-intensity traces. We then group objects in traces by their allocation site. We use per-object hotness and risk to classify allocation sites as *HBM* to *DRAM*, based on heuristics. This classification constitutes advice which we use to annotate allocation sites as *HBM* in Java bytecode. All other allocation sites default to *DRAM*. During production, RiskRelief uses a unique allocation sequence for HBM-marked allocation sites. This sequence places hot and low-risk objects in HBM.

3.5.2 Profiling

RiskRelief relies on offline profiling of Java programs to discover hot and low-risk objects. The outcome of profiling is an *access* trace of per-object reads and writes, see Figure 3.4 for an example (we will discuss the example in more detail later). We track reads and writes in an architecture-independent manner, i.e., we count all load/store accesses to an object's fields. We count accesses to an object's primitive and reference fields, and to its meta-data header, which contains information such as the class type information, synchronization bits, and garbage collector bits.

Profiling per-object accesses can be done in two ways: (1) using read and write barriers in the managed runtime, or (2) using dynamic instrumentation. All generational garbage collectors use reference write barriers for correctness. Write barriers record all mature-to-nursery pointers in a remembered set, which are processed during a minor collection to precisely identify all live nursery survivors. Primitive write barriers are a straightforward extension of reference write barriers. Unlike write barriers, read barriers incur prohibitive overheads [115]. Most production JVMs include collectors that do not require read barriers. Jikes RVM provides both primitive and reference write barriers [5], but does not implement read barriers.

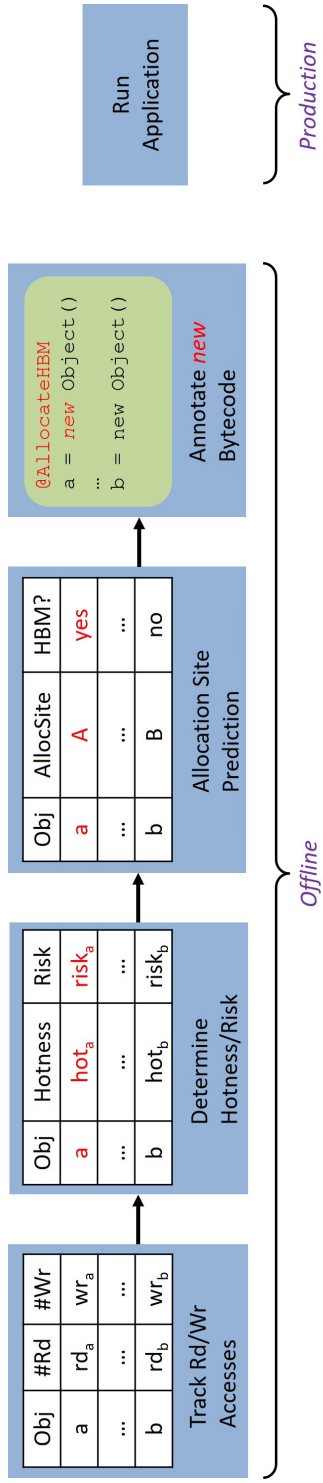


Figure 3.3: Overview of RiskRelief. *Offline analysis records the number of reads and writes to all objects. Then, per-object hotness and risk metrics are used to generate an allocation site classification advice which serves as input to a bytecode rewriter. The rewriter annotates hot and low-risk sites as HBM, steering the garbage collector to place objects in HBM.*

Object	Reads	Writes	Method:Idx
O1	16	8	A():10
O2	16	4	A():10
O3	16	0	A():10
O4	4	8	B():14
O5	4	4	B():14
O6	1	0	B():14

(a) Example access trace

Hotness	AVF-X
24	4
20	1
16	0
12	16
8	4
1	0

(b) Hotness and risk calculation

Heuristic	θ_h	θ_t	θ_{hot}	$\theta_{avf\text{-}x}$	HBM Sites
FMID	1%	---	14	4	A
MRAT	1%	20%	24	16	None
MRAT	1%	40%	20	4	A
MRAT	1%	60%	16	4	A
MRAT	1%	80%	12	1	A & B
MRAT	1%	100%	1	0	A & B

(c) Allocation site prediction

Figure 3.4: Example of an access trace with allocation sites in the last column (a), per object hotness and AVF-X (b), and prediction of allocation sites using the FMID and MRAT heuristics (c).

We therefore rely on dynamic binary instrumentation instead using Pin [113]. Because Pin has no notion of an object's boundary in memory, we deploy a cooperative scheme in which Jikes RVM records each object's starting address, its size in bytes, and its allocation site identifier; in turn, Pin records the number of read and write accesses to each memory location. At the end of the program execution, we gather logs from Jikes RVM and Pin, and we aggregate the two logs to create the access trace which contains all the objects instantiated by each allocation site and the total number of accesses to each object on a per allocation-site basis.

To give each object a unique address in the access trace, we size the mature heap during profiling to preclude full-heap collections. We further set the nursery size to 4 MB. Using this nursery size is a good balance between the size of the access trace and the coverage of mature object behaviors. We label allocation sites with unique identifiers, as in [78].

3.5.3 Allocation Site Classification

After profiling, we analyze the access trace to generate allocation advice, classifying allocation sites as *HBM* versus *DRAM*. Figure 3.4(a) shows an example access trace. Two allocation sites contained in methods A() and B() allocate a total of six objects. The trace also shows the different number of reads and writes to objects. We analyze the trace to compute per-object hotness and risk using the definitions described in the previous sections, see Figure 3.4(b). Next, we use two criteria to label allocation sites: (1) the fraction of total objects allocated from a site that are hot and low-risk, and (2) heuristics to decide which objects are hot and low-risk. If the fraction of hot and low-risk objects allocated from a site is larger than the homogeneity threshold (θ_h), the site is labeled as *HBM*; otherwise, the site is a *DRAM* site. Next, we use two heuristics to qualify objects as hot versus cold, and low- versus high-risk.

Fixed-Midpoint (FMID) is inspired by Gupta et al. [69] and uses the average hotness (or AVF-X) across all mature space objects as the cut-off to quantify

the hotness (or risk) of objects from an allocation site. Specifically, with FMID, we qualify an object as hot if the sum of reads and writes to that object are above the cut-off (average). FMID has the advantage that hotness and AVF-X are straightforward to compute. The disadvantage is that it uses a single cut-off value, which leads to a specific design point in terms of SER, performance and HBM usage. In practice, a heuristic that exposes a trade-off is more desirable, which we advocate in this chapter.

Moving-Ratio (MRAT) uses a ratio namely θ_t (e.g., top-10%) to divide objects into two quadrants, e.g., hot and cold. The hotness cut-off (θ_{hot}) places an object allocated from a site in the top-10% of hot objects. Similarly for identifying low-risk objects, the risk cut-off (θ_{avf-x}) places an object within the top-10% low-risk objects. The user or system administrator specifies the ratio based on environmental constraints. Varying the ratio opens up a trade-off between HBM capacity, performance, and overall SER.

3.5.3.0.1 Example. Figure 3.4(a) shows an example access trace consisting of 6 objects from two allocation sites in methods A() and B(), respectively. Per-object hotness and risk is shown in Figure 3.4(b). We analyze the trace using the FMID and MRAT heuristics, and identify which of the two sites are classified as *HBM* in Figure 3.4(c). We fix θ_h at 1%, and vary θ_t from 20% to 100% for MRAT. The average hotness and risk is 14 and 4, respectively. Therefore, with FMID, the allocation site in method A() has one object (O1) with hotness larger than the average value, and risk larger than or equal to the average risk. Since 1 out of 3 objects from this site are hot and low-risk, which is higher than the homogeneity-threshold of 1%, this site is classified as HBM. Next, we set θ_t to 20% for MRAT and compute the HBM sites. Since θ_t is 20%, we only consider the hottest object (1 out of 6), and the lowest risk object to compute θ_{hot} and θ_{avf-x}. O1 is the hottest leading to θ_{hot} of 24. Similarly, O4 has the lowest risk, leading to a θ_{avf-x} of 16. Neither allocation site in Figure 3.4(a) has an object with both hotness larger than or equal to 24, and risk larger than or equal to 16. Thus, using MRAT with θ_t at 20% leads to all allocations in DRAM. On the other hand, setting θ_t to 40% or 60% results in allocation in HBM for A(). Finally, setting θ_t to 80% and 100% results in all allocations in HBM. This example demonstrates the flexibility exposed by MRAT in exploiting the rich trade-offs that exist between SER, performance, and HBM capacity.

3.5.4 Bytecode Generation

The previous step generates allocation site advice as a file of <site-string, advice> pairs. The advice file only includes the *HBM*-labeled allocation sites. Unlabeled allocation sites default to *DRAM*. Since a minority of allocation sites are labeled *HBM*, 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 a newly introduced new_hbm bytecode. The runtime, when interpreting or compiling the new bytecode, uses the default allocator, called ALLOC_DEFAULT. The runtime then copies all objects allocated by such sites to DRAM if they survive a nursery collection. For the new_hbm bytecode, the runtime uses the newly added ALLOC_HBM allocator. This allocator sets a bit in the object header which notifies the garbage collector to copy these objects to HBM if they survive a nursery collection.

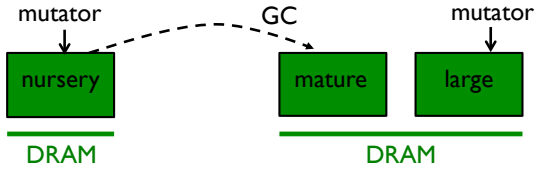
Note that because RiskRelief is a profile-based approach, there might exist allocation sites that were not seen during profiling, i.e., an allocation site was not executed in the profile run while it gets executed in a production run. These unprofiled sites will be unlabeled, and default to *DRAM*, following the above procedure. Future work may explore whether labeling unprofiled sites as *HBM* might be desirable, or whether dynamically profiling just these objects might be tractable and beneficial.

3.5.5 Heap Organization

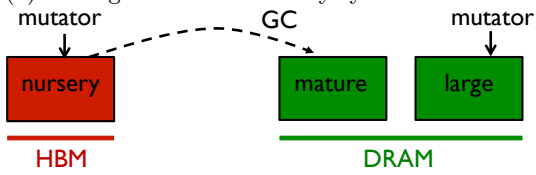
We now describe RiskRelief's heap organizations. The heap organization for a conventional homogeneous DRAM-Only system is shown in Figure 3.5(a). The RiskRelief collectors place the nursery in HBM because the nursery is highly mutated, and hence contains objects that are both hot and low-risk. RR-N places only the nursery in HBM and the rest, i.e., the mature space and large object space, in DRAM, see Figure 3.5(b). RR-M further partitions the mature and large object spaces into DRAM and HBM regions, see Figure 3.5(c).

RR-N operates as follows. Nursery objects are allocated in the HBM nursery. Objects that survive a nursery collection are copied to the mature space in DRAM. Large objects (larger than 8 KB as in our baseline configuration) are allocated directly in the Large Object Space (LOS) which is mapped in DRAM.

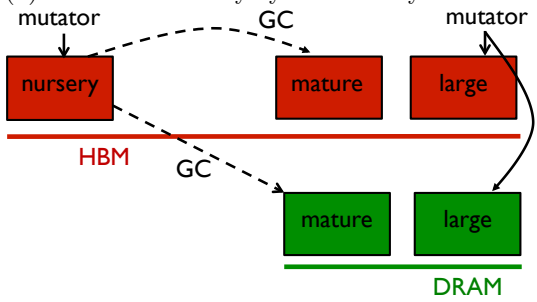
RR-M is more complicated as it requires adjusting the allocation process. In general, new allocation is a two-step process: (1) reserving space and (2) initializing the object header, called post-allocation. For RR-M, post-allocation sets a bit in the object's header if its allocation site is labeled *HBM*, as shown in Figure 3.6. We steal a bit, not in use from the object header in Jikes RVM, and call it the HBM_BIT. Objects with the HBM_BIT set are predicted to be hot and low-risk. During nursery collection, the garbage collector checks the HBM_BIT of each object. If the bit is set, it promotes the object to the mature space in HBM. Otherwise, it promotes the object to the DRAM mature space.



(a) Homogeneous DRAM-only system



(b) RiskRelief-Nursery hybrid memory



(c) RiskRelief-Mature hybrid memory

Figure 3.5: Main memory heap organizations.

```
@Inline
public Address postAlloc(ObjectReference ref, int allocator) {
  if (allocator == Gen.ALLOC_HBM) {
    byte old = readHeaderByte(ref);
    writeHeaderByte(ref, (byte) (old | HBM_BIT));
  }
}
```

Listing 3.6: Our post-allocation sequence sets the HBM_BIT in the header of (predicted) hot and low-risk objects.

RR-M also involves changes to how large objects are treated. For these objects, RR-M's ALLOC_DEFAULT allocates the object directly in the LOS DRAM space, whereas ALLOC_HBM places the object directly in the LOS HBM space.

3.6 Experimental Setup

The main results presented in Section 3.7 are obtained through detailed architectural simulation to accurately assess performance and reliability. This section elaborates on this methodology. We complement these simulation results with emulation results on real hardware in Section 3.8.

Java Virtual Machine and workloads. We use Jikes RVM 3.1.2 [10, 11] and nine applications from the DaCapo suite [29] that work with our simulation and VM infrastructure. We use four benchmarks from the DaCapo-9.12-bach benchmark suite (sunflow, lusearch, pmd, and xalan). We use an updated version of lusearch, called lu.Fix [189], that eliminates useless allocation, and an updated version of pmd, called pmd.S [52], that eliminates a scaling bottleneck due to a large input file. We use three benchmarks from DaCapo 2006: fop, antlr and bloat. As in established methodology, we use 2× the minimum heap size for our benchmarks, and we use different inputs for profiling (default) versus measurement (large). We consider 32-instance workloads of our benchmarks to generate realistic memory traffic.

Measurement Methodology. We follow best practices in Java performance evaluation [30, 71, 79]. We use replay compilation to eliminate non-determinism introduced by just-in-time compilation. During a profiling run, the VM records a plan with the optimization level for each method for the run with the shortest execution time. We then run each benchmark for two iterations. In the first unmeasured iteration, the JIT compiler applies the optimization plan to each method. We then measure the second iteration, which excludes compilation overhead and which represents application steady-state behavior. We report the average across four simulation runs.

Simulator. We use Sniper [36] v6.0, a parallel and high-speed cycle-level x86 simulator for multicore systems, using its most detailed cycle-level hardware-validated core model. Prior work extended Sniper for managed language runtimes, including dynamic compilation, and emulation of frequently-used system calls [158].

Simulated architectures. We consider a 32-core processor with three memory systems: DRAM-Only, HBM-Only (both with 32 GB of main memory) and a hybrid HBM-DRAM system with 2 GB HBM and 32 GB DRAM, see also Table 3.1. We emphasize that the 32 GB HBM-Only system is an idealized but unrealistic point of comparison. We further assume a shared 32 MB L3 cache, 25.6 GB/s DRAM bandwidth and 128 GB/s HBM bandwidth. We assume SEC-DED ECC for HBM because of its lower complexity and power consumption [133, 136]. In line with production systems, we assume single-Chipkill ECC for DRAM.

Simulating multi-programmed Java workloads is time-consuming. Specifically, simulating a 32-core system executing 32 instances of the same Java benchmark in rate mode takes up to one month of simulation time for several

Processors	**Parameters**
Number of cores	1 socket, 32 cores
Core frequency	4.0 GHz
Issue width	4-wide out-of-order
ROB size	128 entries
Branch predictor	hybrid local/global predictor
Caches	**Parameters**
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 32 MB, 16 way, 30 cycle
HBM	**Parameters**
Capacity	2 GB for hybrid, 32 GB for HBM-only
Bus frequency	500 MHz (DDR 1.0 GHz)
Bus width	128 bits
Channels	8 channels
Banks	8 banks/channel
ECC	SEC-DED ECC [76]
tCAS-tRCD-tRP-tRAS	45-45-45-180 CPU cycles
DRAM	**Parameters**
Capacity	32 GB
Bus frequency	800 MHz (DDR 1.6 GHz)
Bus width	64 bits
Channels	2 channels
Banks	8 banks/channel
ECC	single-ChipKill ECC [49]
tCAS-tRCD-tRP-tRAS	45-45-45-180 CPU cycles

Table 3.1: Simulated system parameters.

benchmarks. Moreover, we ran into simulator infrastructure issues when simulating that many cores. We therefore report results for a single-core system with all shared hardware structures scaled down proportionally. We simulate an L3 cache of 1 MB/core, DRAM bandwidth of 0.8 GB/s for each core, and HBM bandwidth of 4 GB/s per core.

Our analysis confirms that the reported experimental results are conservative – in reality, the improvements in SER are almost the same while the improvements in performance are much higher. Figure 3.7 reports the soft error rates for the RiskRelief collectors and a DRAM-only system compared to a HBM-only system. These experimental results are collected from the single-core and 4-core system simulations, in which the shared resources are scaled using the same methodology described above. We find that the SER improvement for a hybrid memory system with the proposed RiskRelief collectors based on the single-core system simulations is nearly the same with the improvement obtained from the 4-core simulations. We thus expect that the SER improvement based on the single-core simulations provides a convincing quantification of the expected SER improvement on the target system. The performance benefit obtained from the target system is expected to be much higher than the benefit as suggested from the single-core simulations, which will be confirmed and analyzed in detail in Chapter 5.

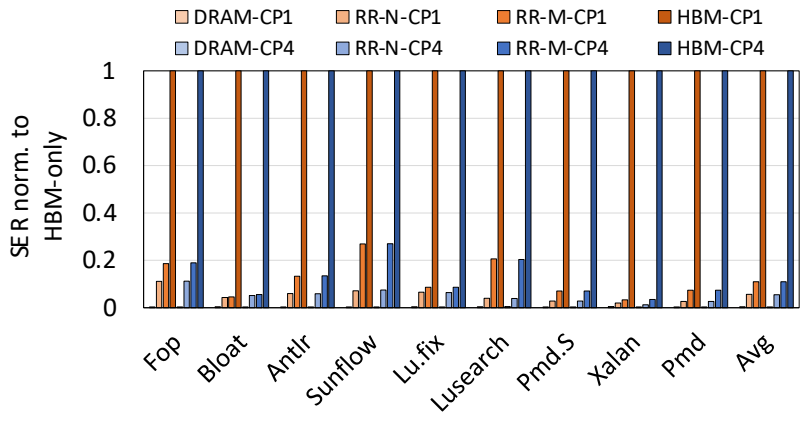


Figure 3.7: Soft error rates normalized to HBM-Only for the RiskRelief collectors and DRAM-Only through single-core and 4-core simulations.

Page migration overhead. We compare RiskRelief to the state-of-art reliability-aware OS approach for hybrid memories proposed by Gupta et al. [69]. Page migration overhead is critical to such OS approaches and includes (1) the latency for moving pages between HBM and DRAM, and vice versa, and (2) TLB shootdown overhead.[1] We assume the latency of copying pages across DRAM and HBM to be 5,000 CPU cycles, which is in line with prior work [19, 60]. We believe this is an optimistic assumption in favor of the OS approach given the available bandwidth in the DRAM and HBM memory subsystems — note that this only affects the OS approach and not RiskRelief because the latter does not migrate pages but objects between the DRAM and HBM memories. The total overhead of a TLB shootdown is independent of the application and depends on the number of cores in the system. The OS keeps track of the 'slave' cores that requested a modified virtual to physical page mapping in the past. During a TLB shootdown, the 'initiator' core requests all slave cores to invalidate the modified TLB entries, flushes its own TLB and waits for the responses from all the slave cores. Following prior work by Villavieja et al. [181], we model the overhead of a TLB shootdown in a system with N cores as follows:

$$T_{shootdown} = N \times T_{slave} + T_{initiator},$$

with T_{slave} and $T_{initiator}$ the time overheads incurred by each slave and initiator cores, respectively. We use published overhead numbers [60] scaled to our 4 GHz processor.

SER calculation. SER, as mentioned before, is computed as the device's raw FIT-Rate times its AVF. We use the default configuration of FaultSim

[1] Gupta et al. [69] account for the page migration overhead but not the TLB shootdown overhead.

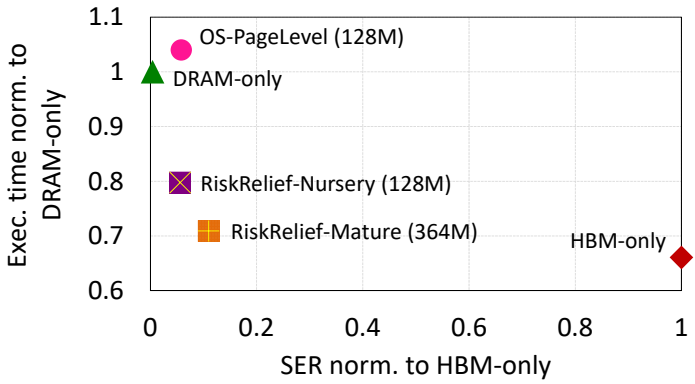


Figure 3.8: The execution time versus SER trade-off for the RiskRelief collectors and the state-of-the-art OS approach, normalized to the DRAM-Only and HBM-Only systems. RiskRelief-Nursery and OS approach consume 128 MB HBM. RiskRelief-Mature uses a larger fraction of HBM (364 MB) by placing part of the mature space in HBM as well.

for evaluating our hybrid HBM-DRAM architecture [130]. FaultSim's default transient FIT rate values for DRAM and HBM are based on a field study conducted on the Oak Ridge 'Jaguar' supercomputer [170]. We further assume SEC-DED and single-Chipkill ECC for HBM and DRAM, respectively. Using this methodology, we find that the FIT-Rates equal 0.1140 and 0.0005 for HBM and DRAM, respectively. Our simulation platform precisely computes AVF by counting the number of reads and writes per cache line, which is not possible on real hardware. More specifically, we logically divide memory into 64-byte cache lines and measure the number of reads and writes per cache line, which we then use to compute AVF per cache line. For a hybrid HBM-DRAM system, we first compute the SER for DRAM and HBM as the product of their respective FIT-Rate and AVF. We then scale the individual SER numbers by the percentage of program heap that is placed in DRAM and HBM.

3.7 Results

We now evaluate RiskRelief collectors across three primary metrics: (1) SER, (2) performance and (3) HBM capacity. Unless otherwise stated, we set θ_h to 1% and θ_t to 20%.

3.7.1 Key Trade-Offs

Using HBM to store a portion of the program heap provides a reliability/performance trade-off, see Figure 3.8. An HBM-Only system delivers the best performance, but the heap is highly susceptible to soft errors, i.e., the

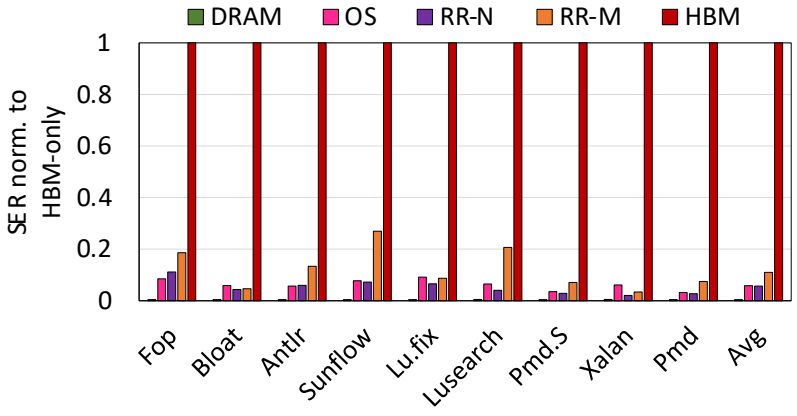


Figure 3.9: Soft error rates normalized to HBM-Only for the RiskRelief collectors, the OS approach and DRAM-Only.

overall normalized SER equals 1. On the other hand, a DRAM-Only system is 34% slower than HBM-Only, but SER is close to 0 (0.003 to be precise). RiskRelief-Nursery places the nursery in HBM and achieves 20% higher performance than a DRAM-Only system. It also reduces SER by 18× compared to an HBM-Only system. HBM capacity for the 32-core system equals 128 MB, which is moderate relative to the total 2 GB HBM capacity.

The state-of-the-art OS approach achieves roughly similar SER as RiskRelief-Nursery, while also requiring 128 MB HBM capacity. Our analysis shows that the OS approach correctly predicts that the nursery is hot and low risk. It thus migrates the nursery pages to HBM. Unfortunately, on x86 multi-core platforms, page migrations incur a substantial performance penalty. The large number of page migrations results in high overhead, and the OS approach performs 24% worse than RiskRelief-Nursery. The significant performance penalty of the OS approach makes it unsuitable for Java applications because the benefits of high HBM bandwidth to access highly mutated and frequently read data is offset by the high cost of page migrations. Our analysis further shows that the overhead of TLB shootdowns is the major contributor to the high cost of page migrations.

Both the state-of-the-art OS approach and RiskRelief-Nursery place the nursery in HBM, using only a modest fraction of the available HBM capacity. Figure 3.8 shows that the RiskRelief-Mature collector uses a larger fraction of the available HBM capacity by placing part of the mature heap space in HBM as well. RiskRelief-Mature is highly effective at improving performance beyond RiskRelief-Nursery. On average, the execution time reduces by an additional 9% compared to RiskRelief-Nursery, and by 29% compared to a DRAM-Only system, while still improving SER by a factor 9× compared to an HBM-Only system.

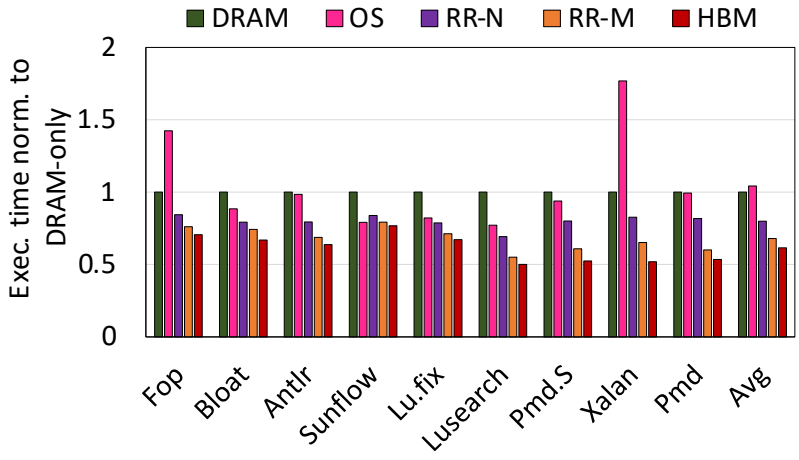


Figure 3.10: Execution times normalized to DRAM-Only for the RiskRelief collectors, the OS approach and HBM-Only.

3.7.2 Soft Error Rate

We now discuss soft error rates for the different systems we evaluate in this work. Figure 3.9 shows SER of DRAM-Only, RR-N, RR-M, and the OS approach, normalized to HBM-Only for the individual workloads. We observe that a DRAM-Only memory system is highly reliable with negligible SER compared to HBM-Only. This observation is consistent with prior work which reports that in DRAM-Only systems, non-DRAM failures, such as those in memory controllers and memory channels, dominate the majority of errors [117]. Whereas HBM-Only is highly unreliable with a normalized SER of 1, RR-N reduces the SER by 18× on average. All benchmarks observe a reduction in SER and the reduction in SER varies from 9× (Fop) to 48× (Xalan). The differences in per-benchmark SER reduction are due to access patterns in the nursery, more specifically, the ratio of nursery writes to reads. RR-N is the most reliable system of all systems we evaluate in this work, but it does not fully utilize the available HBM capacity. We can utilize the available HBM capacity to gain more performance. As mentioned before, RR-M is the best performing system, however, it sacrifices reliability over RR-N. Still, RR-M reduces SER by 9× over HBM-Only. Some benchmarks, such as Bloat, experience no change in SER reduction with RR-M compared to RR-N. This phenomenon occurs because SER depends on several factors including the ratio of object writes to reads, the rate of memory allocation, and how often the objects in the program heap are accessed after the first allocation. Per-benchmark SER reduction with RR-M compared to HBM-Only varies from 5× to 30×. For completeness, the OS approach achieves a normalized SER that is comparable to RR-N.

3.7.3 Performance

We show per-benchmark performance results in Figure 3.10, normalized to a DRAM-Only system. Execution time with an HBM-Only system reduces by 34% on average. Individual benchmarks show a variety of trends. For example, the execution time of Xalan, Pmd, Pmd.S and Lusearch reduces by more than 40%. As reported in Table 3.3, these benchmarks are characterized by either large heaps, high allocation rates, or high nursery survival rates. The compute-bound Sunflow benefits the least from HBM bandwidth. Our analysis indicates that memory read operations in Sunflow exhibit very high on-chip cache hit rates, thus leading to limited traffic to main memory.

The RiskRelief collectors deliver performance in-between DRAM-Only and HBM-Only. Placing the nursery in HBM with RiskRelief-Nursery (RR-N) reduces execution time by 20% on average compared to a DRAM-Only system. Benchmarks that allocate rapidly benefit more from HBM bandwidth. For example, Lusearch allocates the largest volume of objects across our benchmarks, and RR-N reduces its execution time by 34%. The reasons for this large reduction in execution time include: (1) faster read and write operations to memory, (2) higher throughput of memory zeroing to provide security as guaranteed by Java semantics [1, 189], and (3) faster nursery collections. Surprisingly, Sunflow allocates young objects rapidly in the nursery and has the largest number of nursery collections of all of our benchmarks, yet its execution time reduction with RR-N is only 15%. This small reduction is because Sunflow has a small nursery survival rate (only 2%) and copying nursery survivors to the mature space does not require high bandwidth. RiskRelief-Mature (RR-M) reduces the execution time on average by an additional 9% over RR-N, and by 29% over a DRAM-Only system. RR-M splits the mature and large object spaces across DRAM and HBM. The benchmark that benefits the most from RR-M is Lusearch. The execution time of Lusearch reduces by 43%. The performance of Lusearch with RR-M is only 5% less compared to HBM-Only, showing the effectiveness of RR-M in exploiting HBM's high bandwidth. Similarly, the performance of Xalan and the two variants of Pmd also improve substantially with RR-M.

The OS approach leads to a significant performance degradation compared to RR-N. Performance degrades for most benchmarks and we note a significant performance degradation for Fop (42%) and Xalan (77%). The reason is the high number of page migrations per unit of time, see also Table 3.2 which reports the number of page migrations from DRAM to HBM and vice versa, the number of 100 ms migration epochs, and the number of page migrations per epoch. We note that Fop and Xalan are the benchmarks with the highest number of page migrations per unit of time: 347.3 and 463.5 migrations per epoch. Each page migration incurs the overhead of copying the pages and TLB shootdowns. Our measurements indicate that TLB shootdowns account for 41% and 45% of the total execution time for Fop and Xalan, respectively. In other words, TLB shootdowns lead to significant performance degradations for workloads that

	Migrated Pages			Migration	Migrated Pages
	DRAM→HBM	HBM→DRAM	Total	Epochs	/Epoch
Fop	1037	5	1042	3	347.3
Bloat	1921	494	2415	33	73.2
Antlr	1038	12	1050	7	150.0
Sunflow	1574	556	2130	66	32.3
Lu.fix	1323	23	1346	26	51.8
Lusearch	3584	1522	5106	76	67.2
Pmd.S	1083	42	1125	10	112.5
Xalan	23011	3406	26417	57	463.5
Pmd	2263	111	2374	18	131.9
Avg	4093	686	4779	33	158.8

Table 3.2: The number of page migrations (DRAM to HBM, HBM to DRAM, and total), the number of 100 ms migration epochs, and the number of page migrations per epoch for the OS approach.

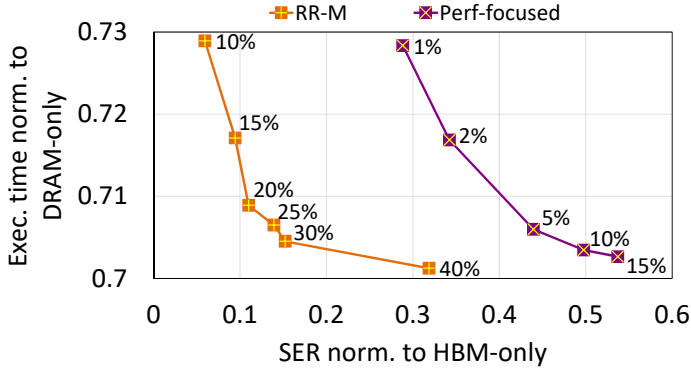


Figure 3.11: Execution time versus SER trade-off for different configurations of RR-M and its performance-focused variant.

incur a large number of page migrations per unit of time. The OS approach delivers performance that is better than RR-N for Sunflow, and only slightly worse than RR-N for Bloat, Lu.fix and Lusearch. This is due to the relatively small number of page migrations per 100 ms epoch for these benchmarks, see Table 3.2. The number of page migrations per epoch is substantially smaller for these benchmarks — Sunflow (32.3), Bloat (73.2), Lu.fix (51.8) and Lusearch (67.2) — compared to the other benchmarks with more than one hundred and up to several hundreds of page migrations per epoch; note that Sunflow has the lowest number of page migrations which leads to a small performance overhead (6%) and a net performance improvement over RR-N.

	Allocation	Heap	RR nursery	RR-M-10%		RR-M-20%		RR-M-30%		RR-M-40%		OS-PageLevel	
	MB	MB	survival %	avg	max	avg	max	avg	max	avg	max	avg	max
Fop	1792	2560	20%	169	184	215	269	230	296	235	302	106	129
Bloat	39872	2112	4%	162	180	167	190	167	187	185	214	128	168
Antlr	7872	1536	15%	250	341	276	393	324	491	339	518	128	128
Sunflow	61440	3456	2%	190	245	410	707	479	850	484	860	124	127
Lu.fix	27136	2176	2%	177	217	177	215	177	216	191	244	126	156
Lusearch	137408	2176	4%	972	1478	959	1474	966	1487	960	1491	121	156
Pmd.S	6464	3136	27%	177	215	327	520	389	659	418	723	128	129
Xalan	31360	3456	14%	210	278	301	432	320	437	322	435	168	515
Pmd	11648	3136	23%	335	513	449	730	608	1019	677	1150	122	140
Avg	36110	2638	12%	294	406	364	548	407	627	423	660	128	183
Heap %				12%		18%		21%		23%		5%	

Table 3.3: Object demographics: total allocation, heap size, nursery survival rates, and average and maximum mature heap usage (in MB) for our 32-instance workloads.

3.7.4 RR-M versus Performance-Focused GC

Utilizing HBM capacity is a trade-off between performance and reliability. RR-M can be configured in a variety of ways to exploit this trade-off space. Performance improves when RR-M is configured to place more mature-space objects in HBM, but this compromises reliability. We show this trade-off in Figure 3.11. We vary θ_t from 10% to 40%. Execution time reduces by 3%, but the SER increases by 5.4×. The reason for the SER increase is that, as RR-M tries to achieve higher performance by placing an increasingly larger fraction of the mature space in HBM, it copies objects with low AVF to HBM. In other words, as θ_t increases, allocation sites with a larger number of high-risk objects are classified as HBM, which results in higher performance, but lower reliability.

Figure 3.11 plots a similar performance versus reliability trade-off curve for a performance-focused variant of RR-M. This performance-focused variant labels allocation sites as HBM based only on the percentage of hot objects allocated from the site. Similar to RR-M, it uses the θ_t threshold to classify objects as hot versus cold. The resulting trade-off curve with this performance-focused collector clearly shows the benefits of RiskRelief in mitigating HBM's high susceptibility to soft errors. Specifically, for the same performance, RR-M exhibits 4.8× lower SER than the performance-focused variant. RR-M takes into account both how often an object is accessed and its AVF before placing it in HBM.

3.7.5 Memory and Demographic Analysis

Table 3.3 summarizes total allocation, nursery survival rates, and percentage of mature heap in HBM for RR-M for the different 32-instance workloads. Our applications allocate frequently ranging from 1.8 GB (Fop) up to 137 GB (Lusearch). Our nursery survival rates vary from 2% to 27%. Copying objects to HBM is faster than DRAM, and hence benchmarks that copy a larger fraction of objects to HBM on a nursery collection benefit more from HBM's high bandwidth. Examples include Xalan and Pmd. The next columns show the average and maximum HBM (in MB) for different configurations of RR-M. Specifically, we show the HBM capacity in MB for different θ_t thresholds. HBM capacity with the most reliable RR-M configuration (θ_t of 10%) equals 294 MB on average, and up to 972 MB. Lusearch consumes the largest HBM capacity with close to 1.5 GB. RR-M places only 12% of the total heap volume in HBM with a 10% θ_t threshold. The percentage of heap volume in HBM increases to 23% of the total heap volume in HBM with a θ_t of 40%. On the other hand, the OS approach places only 5% of the total heap in HBM.

3.8 Evaluation on Real Hardware

Accurately assessing SER for a hybrid memory systems requires per-cacheline read/write statistics which we can only obtain through simulation. We now complement our simulation results with experimentation on commercial hardware, for three reasons: (1) to demonstrate that we can deploy RiskRelief on real systems, (2) to show that RiskRelief directs the vast majority of writes to HBM, and (3) to report the runtime overhead of RR-M relative to RR-N.

Emulation platform. Since we lack access to a commercial machine with HBM, we emulate hybrid HBM-DRAM memory on an existing multi-socket NUMA platform, as in [6]. Commercial HBM systems present HBM as an additional NUMA node to the OS [141], which is exactly what we emulate. In other words, by running Java workloads on the emulation platform with RiskRelief collectors, we incorporate OS and runtime effects as expected on commercial HBM systems. We isolate the Java workload on one socket and disable the other socket. We populate both sockets with commodity DRAM chips. Local memory emulates HBM, and remote memory emulates DRAM. We modify Jikes' MMTk to split the virtual heap into HBM and DRAM. Our two-socket Intel Sandy Bridge E5-2650L processor has 8 physical cores per socket and two hyperthreads per core. We use Ubuntu 12.04.2 with a 3.16.0 kernel. We run 8-instance workloads to utilize all the available cores.

Number of writes to HBM. We now quantify the number of writes to HBM versus DRAM on the emulation platform which 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 available bandwidth to memory is 51.2 GB/s, more than the maximum bandwidth consumed by any of our workloads. A QPI link that supports up to 8 GB/s connects the two sockets. We use Intel's PCM-memory utility to measure the number of writes to HBM and DRAM.

RiskRelief allocates the frequently accessed low-risk objects in HBM and the rest in DRAM. We thus expect that most writes happen to HBM. We observe that in simulation, on average, 90% and 87% of writes happen to HBM for RR-M and RR-N, respectively. On the emulation platform, we find that 87% and 83% of writes happen to HBM, respectively. Simulation and emulation thus confirm that RiskRelief captures the vast majority of writes to HBM — this indicates that the frequently accessed low-risk objects are indeed allocated in HBM. The small discrepancy between emulation and simulation is a result of differences in the OS, hardware prefetcher, memory controller, among other things.

RR-M runtime overhead. RR-M incurs runtime overhead because of the extra steps involved during post-allocation and nursery evacuation. To quantify these overheads as accurately as possible, we compare the performance of RR-

M versus RR-N on the emulation platform while placing the entire heap on one socket of our NUMA platform. On average, the overhead incurred by RR-M is less than 1%, with a maximum of 1.3% for lusearch.

3.9 Other Related Work

Beyond the related work already discussed in Section 3.2, some prior work focuses on automated memory management for hybrid DRAM-PCM memories. However, to the best of our knowledge, this is the first work to automatically manage memory to improve soft error reliability in 3D-stacked memories by dynamically allocating objects to HBM versus DRAM through garbage collection in the managed language runtime.

Production systems now combine DRAM with non-volatile memory (NVM) to deliver high capacity and performance. The most promising NVM, Phase Change Memory (PCM), suffers from low write endurance. Gao et al. [64] use hardware and OS cooperation to expose defective lines in PCM to the garbage collector to avoid allocation in defective lines.

Write-rationing garbage collection for hybrid DRAM-PCM memories [5] places frequently written objects in DRAM to protect PCM from writes and extend its lifetime. More specifically, Kingsguard-Nursery places the nursery in DRAM because the nursery is highly mutated. Kingsguard-Writers dynamically monitors objects to discover highly written mature objects. Crystal Gazer exploits offline profiling to identify allocation sites that produce highly written objects [7].

Wang et al. [182] focus on Big Data systems (e.g., Spark) and leverage GC to place highly accessed information in DRAM in hybrid DRAM-PCM systems. They exploit memory semantics in the Java runtime and focus solely on performance.

3.10 Conclusion

Emerging high-bandwidth memory (HBM) uses 3D stacking to offer more bandwidth than DRAM. Unfortunately, its capacity is limited, and soft error rate is high. Due to greater bit density and new failure modes, hardware error correction alone is insufficient to make HBM reliable. Prior software approaches that leverage the OS to place hot and low-risk pages in HBM have several drawbacks as they operate at a coarse-grained page granularity and introduce page migration overheads that are prohibitive for multicore systems.

This work explores garbage collection in managed runtimes to balance reliability and performance for a hybrid HBM-DRAM memory system. We propose reliability-aware garbage collection to allocate fine-grained hot and low-risk objects in HBM. Both RiskRelief-Nursery and RiskRelief-Mature place the

nursery for young objects in HBM because the nursery is highly accessed and low-risk. RiskRelief-Mature further uses allocation-site prediction to map hot and low-risk mature objects in HBM. We show that object hotness and risk are weakly correlated. RiskRelief-Mature thus uses heuristics to classify objects as hot *and* low-risk for allocation in HBM. Reliability-aware garbage collection substantially outperforms the state-of-the-art OS approach, substantially improves SER over an HBM-Only system, and significantly improves performance over a DRAM-Only system. This work shows that exposing 3D stacking to language runtimes is a promising avenue for balancing reliability and performance.

Chapter 4

Scale-Model Architectural Simulation

Computer architects extensively use simulation to steer future processor research and development. Simulating large-scale multicore processors is extremely time-consuming and is sometimes impossible because of simulation infrastructure constraints and/or simulation host compute and memory limitations. The most popular solution is sampled simulation but unfortunately, it failed to solve the simulation problem, especially for a large-scale computer system. A straightforward example is the simulation problem that we faced in the evaluation of the proposed reliability-aware garbage collection (described in Chapter 3). We had to resort to the simulation of a single-core model (as a representative of the target system) because the target system was simply too large to be rigorously evaluated with existing simulation technologies.

This chapter proposes scale-model simulation, a novel methodology to predict large-scale multicore system performance. Scale-model simulation first constructs and simulates a scale model of the target system with reduced core count and shared resources. Target system performance is then predicted through machine-learning (ML) based extrapolation techniques. Configuring the scale model (i.e., changing core count while proportionally scaling the shared resources) enables trading off accuracy versus simulation speed. We propose two extrapolation models in this work: ML-based Prediction and ML-based Regression. Both models involve a training and a prediction phase and ML-based regression has an additional regression phase. A performance model is trained during the training phase using the scaled simulation results for both models. The main difference between both methods is that ML-based prediction requires simulation runs of the target system during training while ML-based regression does not. The training phase of ML-based regression only needs simulation results obtained from small-scale multi-core models instead of the target system, which makes it possible for ML-based regression to predict

performance and other evaluation metrics for the target system not simulated for time and/or infrastructure limitations. We believe that scale-model simulation opens up a brand new avenue to predict performance for future large-scale multicore system which speeds up architectural simulation and avoids unpredictable simulation limitations. In this chapter, we evaluate the proposed scale-model simulation in detail using multiprogram SPEC CPU workloads and the evaluation results of Java workloads will be presented in Chapter 5. In the next chapter, we will also apply the simulation and prediction methodology based on scale models to the evaluation of reliability-aware garbage collection, in order to solve the simulation problems aroused in Chapter 3.

Section 4.1 discusses the challenge to predict performance for a large-scale future computer system and introduces the proposed scale-model simulation technique. The first step of scale-model architectural simulation is to construct scale models for the target system where core count and shared resources are reduced proportionally – details are described in Section 4.2. We then propose three extrapolation models in Section 4.3: No Extrapolation, ML-based Prediction and ML-based Regression. The simulation setup and workloads construction are elaborated in Section 4.4. In Section 4.5, we first evaluate scale-model construction and scale-model extrapolation using homogeneous workloads and then report the effectiveness of various prediction models under heterogeneous workload mixes. We also leverage the prediction accuracy and simulation speedup in this Section. The sensitivity analysis in Section 4.6 involves evaluation on memory bandwidth scaling, regression models, ML models inputs, multi-core scale-models under regression, memory bandwidth utilization and multi-threaded workloads. Section 4.7 concludes the prior related work on scale models, performance prediction and simulation speed-up, followed by the conclusion in Section 4.8.

4.1 Introduction

Predicting performance for a future computer system is a challenging and critical problem. The traditional approach is to employ detailed architectural simulation. Unfortunately, simulation is extremely time-consuming. In addition, simulation infrastructures have their limitations and may not be able to simulate a future large-scale system because of excessive memory consumption, simulator infrastructure limitations, or insufficient compute capability and/or memory capacity in the simulation host system when simulating large numbers of cores. Researchers and practitioners employ a variety of techniques to tackle the simulation challenge. A widely used solution is sampled simulation [161, 188]. Unfortunately, this approach does not solve the simulation problem when it comes to simulating increasingly large target systems. In particular, we observe that simulating an 8-core, 16-core and 32-core target system using Sniper [36], a fast and state-of-the-art parallel multicore simulator, takes 8, 17 and 43 hours, respectively, on a powerful 36-core simulation

host when running multiprogram SPEC CPU workloads with (only) one billion instructions per benchmark. The super-linear increase in simulation time and complexity as a function of system size is a major challenge for computer architects in academia and industry.

In this chapter, we propose *scale-model simulation*, a novel paradigm to predict future system performance. Scale-model simulation combines architectural simulation with machine learning to predict performance for large-scale systems based on detailed simulation of a scaled-down configuration of the target system, called the *scale model*. Scale-model simulation first simulates a scale model of the target system. Performance for the target system is then predicted through extrapolation. Scale models solve the two problems aforementioned: (1) scale models speed up the simulation of large-scale systems: scale models are small enough to simulate in reasonable amount of time while performance extrapolation is instantaneous; and (2) scale models make simulation feasible for large-scale systems that cannot be simulated on existing infrastructure because of limitations in memory and compute capacity.

Scale models are widely used in a variety of engineering disciplines, including civil engineering (e.g., construction, fluid dynamics), mechanical engineering (e.g., aerodynamics, engine design), construction (e.g., architectural design, city development), etc. The most familiar scale models are miniatures, i.e., scaled-down versions of an original object. A key property of a scale model is that it accurately maintains relationships between various important aspects, but not necessarily all aspects, of the original object. Scale models enable demonstrating or studying some behavior of the original object. To the best of our knowledge, scale models have not been applied to the field of general-purpose computer architecture. While building an exact miniature of a target system may be hard in the context of processor architectures, if at all possible, we leverage the idea of scale models to predict future computer system performance.

The scale-model simulation paradigm can be decomposed into two subobjectives: (1) scale-model construction and (2) scale-model extrapolation. The first objective relates to how to construct a scale model of a (much) larger target multicore system, so that it takes substantially less time to simulate than the target system, yet enables an accurate prediction of the performance of the large-scale target system. A scale model is a scaled-down version of the target multicore system by featuring a reduced number of cores, say by a factor F, relative to the target system. The question is what to do with the shared resources, in particular the last-level cache (LLC), NoC and memory bandwidth. One option may be to not scale the shared resources. Assuming no shared resource contention, the performance of a single core in the scale model would be similar to the performance of an individual core in the target system. Of course, in reality, the actual performance will be less because of shared resource contention. We find for our suite of SPEC CPU2017 workloads, that not scaling shared resources leads to largely inaccurate scale models with an average 60% prediction error (and up to 94%) for a single-core scale model versus a

32-core target system. The alternative option is to proportionally scale the shared resources. In particular, when scaling the number of cores by a factor F in the scale model relative to the target system, the shared resources are also reduced proportionally by a factor F, i.e., LLC capacity, NoC bisection bandwidth and memory bandwidth are reduced by a factor F. We find that proportional resource scaling leads to substantially more accurate single-core scale models, with an average prediction error of 14.7% and at most 32.2% relative to a 32-core target system.

Because the scale model is not an exact miniature of the target system, the second objective relates to how to extrapolate performance from the scale model to the target system to further improve accuracy. Shared resources lead to a variety of complex interactions at the system level, which the scale models may or may not capture to a sufficient degree. Scale-model extrapolation predicts the impact of contention effects in shared resources on target-system performance based on the simulated scale model. We propose and evaluate two extrapolation methods that leverage Machine Learning (ML) to infer prediction models that predict target-system performance based on scale-model measurements. The two methods are ML-based prediction and ML-based regression. The key difference between both methods is that ML-based regression does not require simulation runs of the target system during training, in contrast to ML-based prediction. ML-based regression can thus be deployed when it is too time-consuming or even impossible to run simulations of the target system. We explore a variety of ML-based scale-model extrapolation techniques, including decision trees, random forest and support vector machines (SVM), and we find that SVM is most accurate. In addition, we evaluate a number of regression methods (linear, power and logarithmic), and find that logarithmic regression is most accurate. Our evaluation using multiprogram SPEC CPU2017 workloads demonstrates the high accuracy of scale-model simulation. Considering a single-core scale model and a 32-core target system, we report that for homogeneous multiprogram workload mixes, SVM-based prediction yields an average prediction error of 6.4% (20.8% max error). SVM-based regression is slightly less accurate as it does not involve target-system simulations during training. SVM-based regression yields an average prediction error of 8.0% (26.4% max error).

Scale-model simulation is more challenging for heterogeneous multiprogram workload mixes because of more diverse interaction and contention effects. Nevertheless, we demonstrate that scale-model simulation is also effective and accurate for heterogeneous workload mixes. We report that SVM-based prediction achieves an average prediction error of 13.2% (max error of 27.5%) for SVM-based prediction, and 15.8% for SVM-based regression (max error of 28.7%).

Scale-model simulation leads to substantial simulation speedups. Training the prediction model is a one-time cost that can be amortized across many predictions. Once the prediction model has been trained, scale-model simulation is fast. It only requires running a simulation of the application of interest on the single-core scale model, which is substantially faster than running a simulation

#cores	LLC	NoC	DRAM
32	32 MB: 32 slices	128 GB/s: 4 CSLs, 32 GB/s per CSL	128 GB/s: 8 MCs, 16 GB/s per MC
16	16 MB: 16 slices	64 GB/s: 4 CSLs, 16 GB/s per CSL	64 GB/s: 4 MCs, 16 GB/s per MC
8	8 MB: 8 slices	32 GB/s: 2 CSLs, 16 GB/s per CSL	32 GB/s: 2 MCs, 16 GB/s per MC
4	4 MB: 4 slices	16 GB/s: 2 CSLs, 8 GB/s per CSL	16 GB/s: 1 MC, 16 GB/s per MC
2	2 MB: 2 slices	8 GB/s: 1 CSL, 8 GB/s per CSL	8 GB/s: 1 MC, 8 GB/s per MC
1	1 MB: 1 slice	4 GB/s: 1 CSL, 4 GB/s per CSL	4 GB/s: 1 MC, 4 GB/s per MC

Table 4.1: Constructing scale models through *Proportional Resource Scaling*: LLC capacity in MB; on-chip interconnection network in GB/s: number of cross-section links (CSLs) and bandwidth per CSL; main memory bandwidth in GB/s: number of memory controllers (MCs) and bandwidth per MC.

of the target system, i.e., in our experimental setup in which we use Sniper [36] on a high-end 36-core Intel PowerEdge R440 server, we find that simulating a single-core scale-model is 28× faster than simulating the 32-core target system.

In summary, we make the following key contributions:

- We propose scale-model simulation, a novel methodology to predict target-system performance based on scale-model performance simulations.
- We find that shared resources are best proportionally scaled in the scale model relative to the target system.
- We demonstrate that extrapolation can significantly improve scale-model prediction accuracy.
- We propose and evaluate two ML-based extrapolation techniques that do or do not rely on target-system simulations during training.
- We evaluate scale-model simulation and demonstrate high accuracy and simulation speed improvements for both homogeneous and heterogeneous multiprogram workload mixes for a 32-core target system based on single-core scale-model simulations.
- We find that ML-based regression is almost equally accurate as ML-based prediction while not requiring target-system simulations during training, making ML-based regression a more practical approach.

4.2 Scale Model Construction

Scale-model architectural simulation involves two key concerns: (1) how to construct the scale models, and (2) how to build an accurate extrapolation model based on the scale model predictions. We discuss the former in this section and the latter in the next section.

A scale model is a scaled-down version of the large-scale target system such that its performance is a (relatively) accurate representation of the target system. More precisely, the scale model needs to be configured such that its per-core performance is similar to per-core performance in the target system. The challenge when constructing scale models for multicore processors is how to deal with shared resources.

One option is to simply scale the number of cores in the scale model while keeping the shared resources as in the target system — we refer to this approach as *No Resource Scaling (NRS)*. For example, a scale model consisting of a single core would have access to the fully sized LLC capacity as well as the same NoC and memory bandwidth as in the target system.

Another, more accurate, option is to proportionally scale the shared resources with core count — we refer to this approach as *Proportional Resource Scaling (PRS)*. The intuition behind PRS is to provide balanced scale models that exhibit similar degrees of resource contention as in the target system. In particular, when scaling the number of cores by a factor F, we scale LLC capacity, NoC bandwidth and memory bandwidth by the same factor F. In other words, we keep LLC capacity per core constant and we keep interconnection and memory bandwidth per core constant. In our setup, we assume 1 MB of LLC per core, 4 GB/s NoC bisection bandwidth per core, and 4 GB/s memory bandwidth per core. See Table 4.1 for how we scale shared resources in our setup. Since we assume a NUCA LLC with a 1 MB slice attached to each core in our setup, we proportionally scale down LLC capacity as we consider fewer cores in the scale model. Scaling bandwidth is more complicated. We scale DRAM bandwidth by changing both the number of memory controllers and bandwidth per memory controller. Starting from the target system, we first scale down the number of memory controllers from 8 (at 32 cores) to 1 (at 4 cores), and then scale down the amount of bandwidth per memory controller. First scaling the number of memory controllers and then scaling bandwidth per memory controller once there is only single memory controller left, enables more accurate scale models compared to first scaling memory bandwidth per memory controller and then scaling the number of memory controllers (as we will quantify in the evaluation section). For the interconnection network, we scale link bandwidth as the number of cross-section links reduces with core count. In particular, scaling down from 32 to 16 cores, the number of cross-section links remains unchanged, hence we have to halve bandwidth per link from 32 GB/s to 16 GB/s. In contrast, when moving from 16 to 8 cores, the number of cross-section links halves from 4 to 2, hence we maintain the per-link bandwidth at 16 GB/s.

4.3 Scale Model Extrapolation

Scale model construction is only a first step. We need scale model extrapolation to yield even more accurate target system performance predictions.

Scale-model extrapolation considers scale-model simulation results to predict target-system performance. We consider two extrapolation models in this work: no extrapolation and ML-based prediction and regression.

4.3.1 No Extrapolation

The simplest way to predict target system performance is to use the per-core performance observed in the scale model as a prediction for per-core performance in the target system. This approach implicitly assumes that the interference observed in the shared resources in the scale model is similar to (or the same as in) the target system. The scale model that we assume is a single-core system with the shared resources proportionally scaled following the PRS approach. The performance measured for this single-core scale model then is the prediction for per-core performance in the target system.

While we primarily focus on a single-core scale model in this work, it might be worth considering a two-core scale model or a four-core scale model (again, with the shared resources proportionally scaled). This typically leads to higher accuracy. On the flip side, simulating a scale model with more cores and larger shared resources takes longer. In other words, increasing the size of the scale model leads to an accuracy versus speed trade-off. The larger the scale model, the higher the accuracy but the longer simulation takes. While we will primarily focus on the results with a single-core scale model — as it yields the highest possible simulation speedup — we will also explore the accuracy versus simulation speed trade-off by considering larger scale models in the results section.

4.3.2 Machine Learning-based Prediction and Regression

Leveraging Machine Learning (ML) enables achieving higher accuracy compared to the No Extrapolation method. We consider two ML-based approaches: *ML-based Prediction* and *ML-based Regression*. Both methods involve a training phase during which a performance model is trained. The training phase incurs a one-time cost. The key difference between both approaches is that ML-based Regression does not require simulation runs of the target system during training, in contrast to ML-based Prediction. This has important implications in practice. In case it is impossible to simulate the target system for some reason (too long simulation time or other simulator limitations), one has to resort to ML-based Regression. Higher accuracy is typically obtained through ML-based Prediction, although that requires access to the target system. We now explain both approaches.

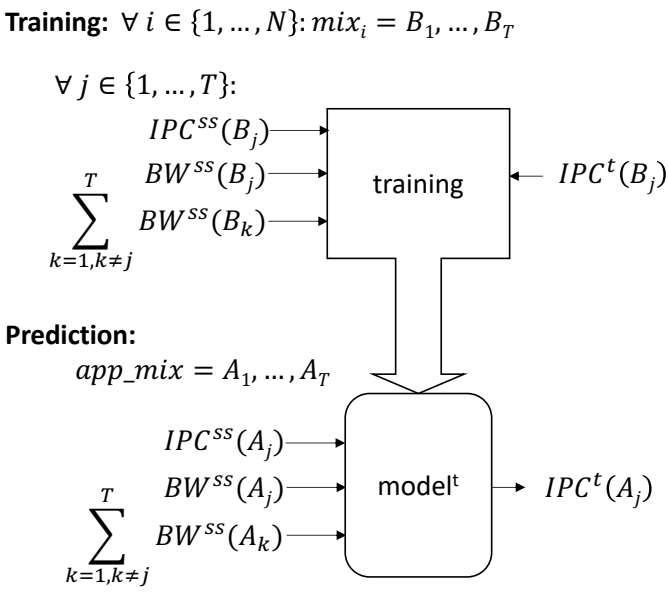


Figure 4.1: ML-based prediction involves a training and prediction phase. *The training phase requires simulation results for the target system.*

4.3.2.1 ML-Based Prediction

ML-based prediction involves a training phase in which a set of training benchmarks are run on both the scale model and the target system, see also Figure 4.1. We consider N benchmark mixes in our training set, with each mix i ($1 \leq i \leq N$) consisting of T benchmarks $B_j, 1 \leq j \leq T$. There are as many benchmarks per mix as there are cores in the target system, namely T. We denote a performance number P obtained on the single-core scale model with superscript ss (P^{ss}), on the multi-core scale models with superscript msX (P^{msX}), and on the target system with superscript t (P^t).

On the single-core scale model, we measure performance (i.e., IPC) and memory bandwidth utilization. The latter is a function of the number of LLC misses per unit of time and has a significant impact on resource contention in the memory subsystem during co-execution with other benchmarks. In other words, it provides a measure for how much contention the particular benchmark is going to create on the shared resources when co-executed with other benchmarks. Our results confirm that considering both performance and memory bandwidth utilization improves accuracy (as we will quantitatively demonstrate in the evaluation section). The performance and bandwidth utilization numbers on the single-core scale model serve as independent variables to the ML technique. More precisely, the input variables to the ML model are per-core performance for each of the benchmarks in the training mix ($IPC^{ss}(B_j)$), alongside

the per-core bandwidth utilization for the given benchmark ($BW^{ss}(B_j)$) as well as the sum of the per-core bandwidth utilization numbers for the co-running applications in the workload mix ($\sum_{k=1,k\neq j}^{T} BW^{ss}(B_k)$). On the target system, we measure performance for each of the benchmarks in the multi-program workload mix ($IPC^{t}(B_j)$). Target system performance for each of the benchmarks in the training mix serves as dependent variables to the ML technique. In other words, the different training samples provide different observations: the independent variables are the IPC and bandwidth utilization on a single-core system, along with the total bandwidth utilization of co-running applications; the dependent variable is the IPC of the target system for the training benchmarks.

Overall, the input to the ML training phase consists of $N \times T$ data points as there are N mixes and T benchmarks per mix. The end result of the training phase is a performance model, denoted as $model^{t}$, that predicts target-system performance of an application when co-run with $T-1$ other applications.

The prediction, or inference, phase involves simulating a previously unseen application A_j (i.e., the workload of interest) on the single-core scale model. The measured performance and bandwidth utilization numbers serve as input to the prediction model which then yields a prediction for performance of the application of interest on the target system. More specifically, the prediction model takes the IPC of the application of interest on the single-core scale model as input ($IPC^{ss}(A_j)$), alongside its bandwidth utilization on the scale model ($BW^{ss}(A_j)$) as well as the total bandwidth consumption of the co-running applications in the workload mix. The latter is computed as the sum of the bandwidth consumption for each of the applications in the workload mix as observed in the single-core scale model, i.e., $\sum_{k=1,k\neq j}^{T} BW^{ss}(A_k)$. The model $model^{t}$ then predicts performance for application A_j on the target system.

We consider different ML techniques in this work to construct the prediction model, namely decision tree (DT), random forest (RF) and support vector machines (SVM) using the scikit-learn v1.0.1 framework.[1] The DT algorithm is an optimized version of the CART (Classification and Regression Trees) algorithm which constructs binary trees by seeking for the largest information gain at each node using Iterative Dichotomiser. RF includes a diverse set of decision trees to avoid overfitting; this is done through two levels of randomization. First, each tree in the ensemble is built for a random subset from the training set. Second, when splitting a node during the construction of a tree, the best split is found for either all input features or for a random subset of features. We use the radial basis function (RBF) as the SVM kernel to capture non-linear performance scaling trends.

[1] `http://scikit-learn.org/`

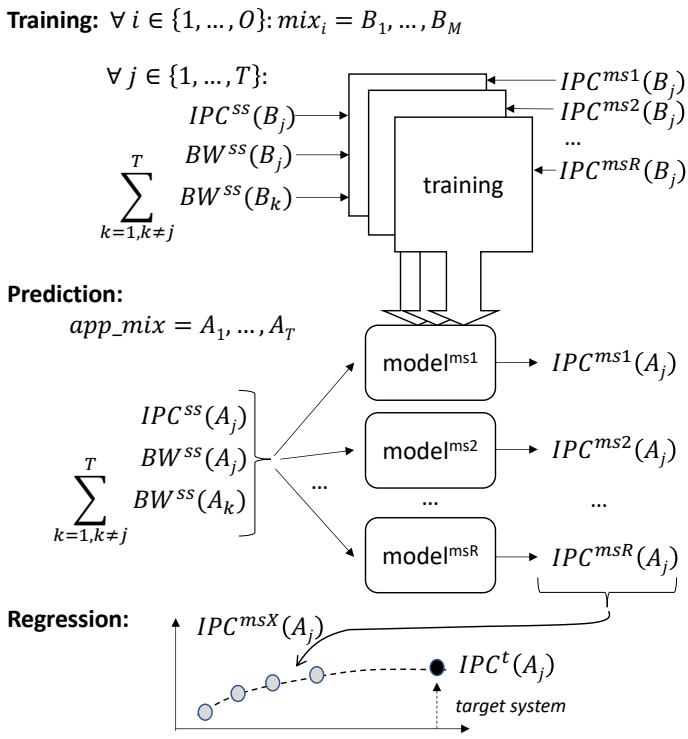


Figure 4.2: ML-based regression involves a training, prediction and regression phase. *The training phase requires simulation results obtained for a number of multi-core scale models, but not the target system.*

4.3.2.2 ML-Based Regression

As mentioned above, ML-based prediction requires simulation runs of the target system during training, which may be a significant impediment in practice. ML-based regression overcomes this drawback by relying on simulation runs of a variety of scale models instead, which is typically easier to achieve in practice. ML-based regression consists of three steps, see also Figure 4.2. In the first step, ML-based regression leverages the ML-based prediction method discussed above to train a number of prediction models. These prediction models do not predict performance for the target system, as under ML-based prediction, but they predict performance for a number of multi-core scale models $ms1, ms2, \ldots, msR$. Note that these scale models feature multiple cores. The training phase involves measuring performance and bandwidth utilization on the single-core scale model, and measuring performance for the multi-core scale models for each of the benchmarks in the training workload mixes. The input to the training phase thus includes, as independent variables, the performance ($IPC^{ss}(B_j)$) and bandwidth utilization ($BW^{ss}(B_j)$)

Processor	
Number of cores	32 cores
Core frequency	4.0 GHz
Issue width	4-wide
ROB size	128 entries
Branch predictor	hybrid local/global predictor
Max. outstanding	48 loads, 32 stores, 10 L1-D misses
Cache Hierarchy	
L1-I	32 KB, 4 way, 4 cycle access time
L1-D	32 KB, 8 way, 4 cycle access time
L2	256 KB per core, 8 way, 8 cycle
LLC	shared 32 MB, 64 way, 30 cycle NUCA, 32 slices, 1 MB/slice, 1 slice/core
NoC	
Mesh topology	4×8
Bandwidth	128 GB/s bisection bandwidth
DRAM	
Memory controllers	8
Bandwidth	128 GB/s aggregate bandwidth

Table 4.2: Target system.

of each benchmark in the mix alongside the aggregate bandwidth utilization of the co-running benchmarks ($\sum_{k=1,k\neq j}^{T} BW^{ss}(B_k)$). The dependent variables are the performance numbers for each benchmark for the scale models $ms1, ms2, \ldots, msR$, or $IPC^{ms1}(B_j), IPC^{ms2}(B_j), \ldots, IPC^{msR}(B_j)$. The ML-based prediction method is used to train the prediction models for the various multi-core scale models.

As a second step, once these prediction models have been trained, we predict performance for a previously unseen application of interest A_j on the multi-core scale models $ms1, ms2, \ldots, msR$. The input to the models includes the application's scale-model performance ($IPC^{ss}(A_j)$), its bandwidth utilization ($BW^{ss}(A_j)$) and the aggregate bandwidth utilization of the co-runners ($\sum_{k=1,k\neq j}^{T} BW^{ss}(A_k)$). The models then predict performance for application A_j on the scale models, namely $IPC^{ms1}(A_j), IPC^{ms2}(A_j), \ldots, IPC^{msR}(A_j)$.

The third step involves regression to predict performance for the target system based on the predicted performance numbers for the multi-core scale models. We consider a number of regression techniques, including linear, power-law and logarithmic regression, to predict target-system performance. We find that logarithmic regression yields the highest accuracy (a quantitative evaluation is reported in the evaluation section).

4.4 Experimental Setup

4.4.1 Simulation Setup

We use Sniper v6.0, a parallel and high-speed cycle-level x86 simulator for multicore systems, using its most detailed cycle-level hardware-validated core model [36]. Our target system is a 32-core processor, see Table 4.2. We simulate 4-wide out-of-order cores with a 3-level cache hierarchy. The LLC is a 32 MB NUCA cache, and we assume a 128 GB/s bisection bandwidth mesh NoC and 128 GB/s main memory system with 8 memory controllers.

Our simulation speed numbers are obtained by running Sniper on a 36-core Intel PowerEdge R440 server. This server is dual-socket machine with 18 cores per socket, 24 MB LLC per socket, 384 GB of memory.

4.4.2 Workloads

We consider both homogeneous and heterogeneous multiprogram workload mixes in the evaluation. The benchmarks are taken from SPEC CPU2017 and we consider 1B-instruction simulation points per benchmark [161]. The homogeneous workloads assume co-running instances of the same benchmark, all starting at (slightly) different offsets. The heterogeneous workload mixes are randomly composed. We finish the simulation and measure performance when the first benchmark in the workload mix has reached the end of its simulation point.

We make sure that the training set is completely disjoint from the evaluation set in all of our experiments. For the homogeneous workload mixes, we use a cross-validation setup in which we use $N-1$ benchmarks for training the models when evaluating prediction accuracy for the Nth benchmark, with $N = 29$ for SPEC CPU2017. There are hence 28 training benchmarks to train a model to predict performance for the 29th benchmark. We use the prediction and extrapolation models to predict performance for the previously unseen application of interest on the target system when co-run with additional $(T-1)$ copies of the application of interest.

For the heterogeneous workload mixes, we consider 8 randomly chosen benchmarks in the evaluation set while using the 21 remaining benchmarks in the training set. The workload mixes in the training set are random mixes. The number of training mixes is chosen such that the total amount of training data is constant, i.e., we consider a total of 320 training results to train an ML model. For ML-based prediction, this means that we consider $N = 10$ training mixes with $T = 32$ benchmarks each, yielding a total of $N \times T = 320$ training results. For ML-based regression, when training an ML model for an M-core scale model, we consider O training mixes, so that we have a total of $O \times M = 320$ training mixes. In particular, when training a model for a two-core scale model, we consider 160 training mixes, yielding 320 training results;

when training a model for a quad-core scale model, we consider 80 training mixes, again yielding 320 training results; etc. Prediction is done for a previously unseen application of interest which we simulate on the single-core scale model. We predict performance for the application of interest on the target system when co-run with 10 random heterogeneous mixes of previously unseen applications from the evaluation set; we report the average prediction error across these 10 mixes for each of application of interest.

4.5 Evaluation

We now evaluate scale model simulation. We first evaluate scale-model construction, and then evaluate scale-model extrapolation. We quantify accuracy using the following absolute prediction error metric: $error = \left|\frac{IPC_{predicted}-IPC_{actual}}{IPC_{actual}}\right|$. IPC_{actual} is the IPC of the application of interest on the target system — in our setup, this is the IPC of a single benchmark instance in a 32-instance multi-program workload. $IPC_{predicted}$ is the predicted IPC of the application of interest on the target system based on measurements obtained through simulation of the scale model. In case of No Extrapolation, the predicted IPC on the target system *is* the IPC obtained on the scale model. In case of ML-based Prediction and Extrapolation, the predicted IPC is provided by the ML model when given the performance metrics for the scale model as input. We assume a single-core scale model in all of our experiments unless mentioned otherwise.

4.5.1 Scale Model Construction

We consider the following four scale-model construction techniques: (1) No Resource Scaling (NRS), i.e., the shared resources in the scale model are sized identically to the target system, (2) Proportional Resource Scaling (PRS) in which we only scale the LLC in the scale model (i.e., DRAM bandwidth in the scale model is the same as in the target system), (3) PRS with scaled DRAM bandwidth only (i.e., LLC capacity is the same in the scale model and target system), and (4) PRS with scaled LLC size *and* DRAM bandwidth. (We evaluated NoC scaling as well but found it to have (virtually) no effect for the workloads considered in this work, hence we exclude it from the discussion.)

Figure 4.3 reports prediction error for the single-core scale model, i.e., we consider a scale model with a single core to predict per-core performance in the 32-core target system. The benchmarks are sorted by their LLC MPKI from left to right. The benchmarks on the left-hand side are thus compute-intensive for which NRS and PRS perform equally well. However, memory-intensive benchmarks appearing on the right-hand side experience contention in the shared resources and hence require that the scale models feature proportionally scaled-down shared resources. Overall, NRS is generally inaccurate

with an average absolute error of 60% and up to 94%. PRS is more accurate, especially for memory-intensive workloads: scaling the LLC brings down the average error to 51.3%, while scaling DRAM bandwidth reduces the average error to 40.5%. Scaling both LLC capacity and DRAM bandwidth has synergistic effects, bringing down the prediction error to 14.7% on average and at most 32.2% (milc). Proportionally scaling all shared resources leads to a scale model that is a relatively accurate representation for per-core performance in the target system.

4.5.2 Scale Model Extrapolation

While PRS leads to relatively accurate scale models, we can do even better through scale model extrapolation. No Extrapolation uses performance obtained for the scale model as a prediction for per-core performance in the target system — this is effectively PRS with scaled resources from the previous section. We further consider ML-based Prediction and ML-based Regression; we consider three ML techniques — Decision Tree (DT), Random Forest (RF) and Support Vector Machines (SVM) — and we use logarithmic regression for the Regression approach.

Figure 4.4 reports the prediction error for these techniques assuming homogeneous workload mixes. ML-based Prediction brings down the average absolute prediction error by a significant margin compared to No Extrapolation (average error of 14.7% and up to 32.2%). SVM is the most accurate ML-based Prediction technique with an average error of 6.4% (maximum error of 20.8%). DT yields an average absolute prediction error of 9.3% (and up to 29.1%), whereas RF leads to an average error of 8.3% (and up to 21.3%). ML-based Regression is slightly less accurate than ML-based Prediction as it does not require simulating the target system during training. Yet, accuracy is still high and SVM with logarithmic regression (SVM-log) yields the highest accuracy among the ML-based Regression techniques with an average absolute prediction error of 8.0% (and at most 26.4%). While ML-based prediction outperforms ML-based regression in general, the inverse is true for some benchmarks. This is the case when performance across scale models (with 2, 4, 8 and 16 cores) follows a predictive trend line — favoring regression. If on the other hand, the relative performance delta between the one-core scale model and the 32-core target system is relatively easy to predict, i.e., the relative delta is fairly similar to previously seen training examples, then prediction is most accurate.

4.5.3 Heterogeneous Workload Mixes

So far, we considered homogeneous workload mixes. Figure 4.5 reports prediction error for the various prediction techniques under heterogeneous workload mixes. The results are consistent with the homogeneous workload mixes, i.e., ML-based Prediction is slightly more accurate than ML-based Regression,

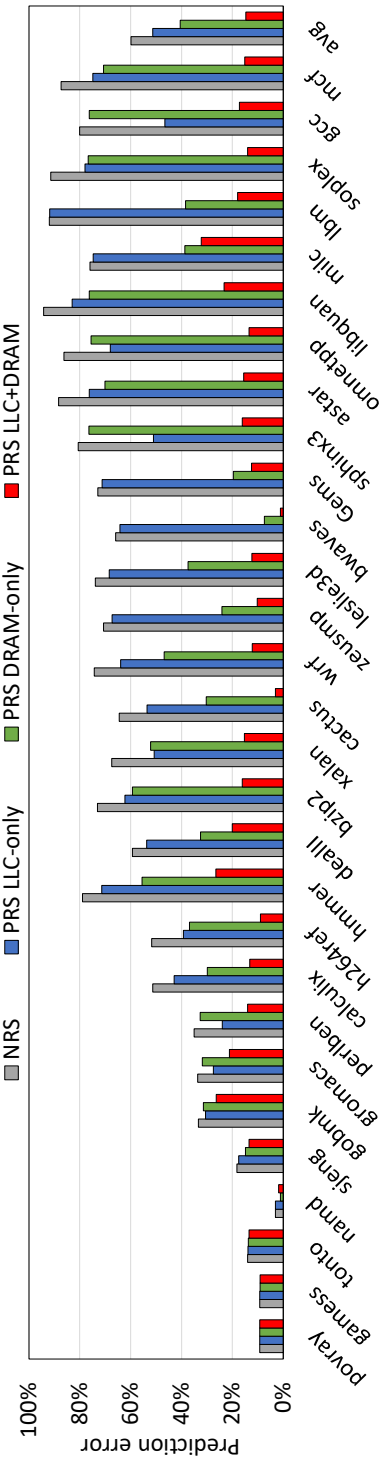


Figure 4.3: Evaluating scale model construction using homogeneous workload mixes: NRS versus PRS with scaled LLC capacity, scaled DRAM bandwidth, and both. *Proportional Resource Scaling (PRS) in which all shared resources are scaled proportionally leads to the most accurate scale models.*

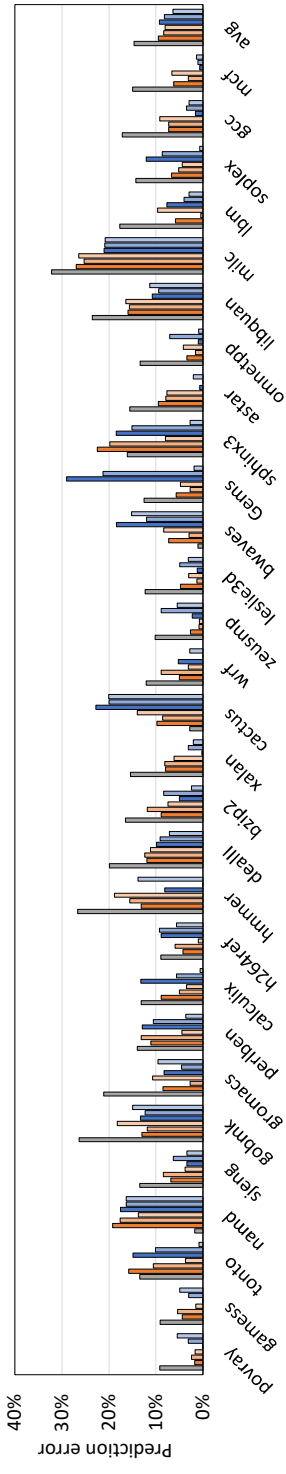


Figure 4.4: Evaluating scale model extrapolation using homogeneous workload mixes: No Extrapolation versus ML-based Prediction (DT, RF and SVM) and Regression (DT-log, RF-log and SVM-log). *SVM-based prediction yields the highest accuracy (6.4% average absolute prediction error), while SVM-based regression (SVM-log) is only slightly less accurate (8.0% average absolute prediction error).*

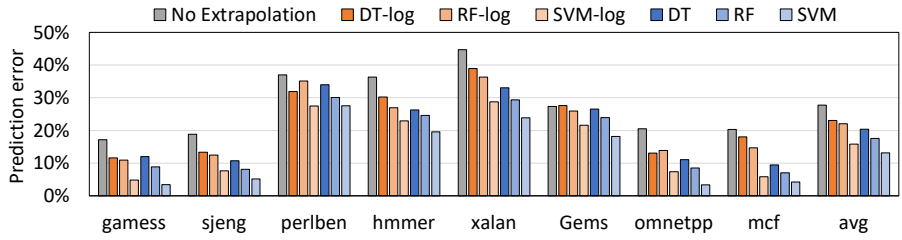


Figure 4.5: Evaluating scale model extrapolation using heterogeneous workload mixes: No Extrapolation versus ML-based Prediction (DT, RF and SVM) and Regression (DT-log, RF-log and SVM-log). *The SVM-based Prediction method yields the highest accuracy (13.2% average prediction error), while SVM-based Regression (SVM-log) is only slightly less accurate (15.8% average prediction error).*

and SVM is the most accurate ML approach. We do note higher prediction errors for the heterogeneous workload mixes compared to the homogeneous workload mixes due to more complex and diverse interactions between co-running applications: average prediction error of 15.8% (max 28.7%) for SVM-log versus 13.2% (max 27.5%) for SVM, versus 27.8% (max 44.7%) for No Extrapolation.

These per-application performance predictions can be used to predict system throughput (STP) on the 32-core target system. STP is computed as the sum of normalized IPC values (IPC on the target system divided by IPC on the single-core scale model) across all applications in the workload mix [56]. Figure 4.6 reports the STP prediction error (sorted) for ML-based regression for a total of 80 heterogeneous mixes. SVM-log is the most accurate regression approach with an average error of 3.8% versus 5.6% for DT-log and RF-log. Interestingly, the STP prediction errors are lower than the per-application prediction errors reported above. The reason is that STP is computed as the sum of normalized IPC values, hence over- and underestimations offset each other.

4.5.4 Simulation Speedup

Scale-model simulation yields a substantial simulation speedup because simulating a scale model takes considerably less time than simulating the target system. And in some cases, it might not even be possible to simulate the target system, due to simulator infrastructure and/or simulation host constraints. Once scale-model simulation results are available, predicting target-system performance is almost instantaneous provided that the ML model has been trained offline.

Figure 4.7 reports prediction error versus simulation speedup compared to simulating the 32-core target system. The No Extrapolation curve consists of 5 data points. The data point on the far right refers to the case where the scale model is a single-core system. Moving to the left, we have a dual-core,

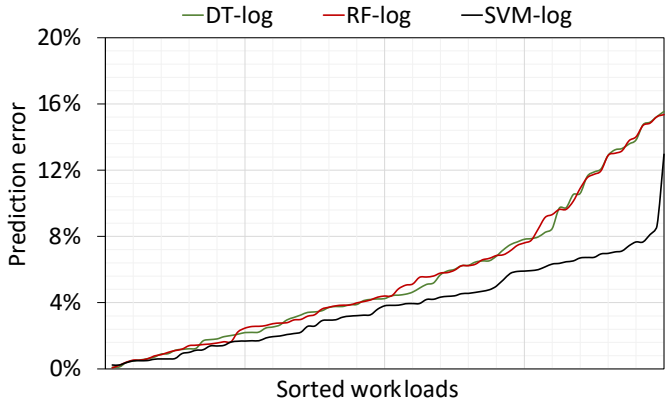


Figure 4.6: STP prediction error for ML-based regression across a total of 80 heterogeneous workload mixes. *SVM-log predicts system throughput (STP) with an average prediction error of 3.8% and at most 13.0%.*

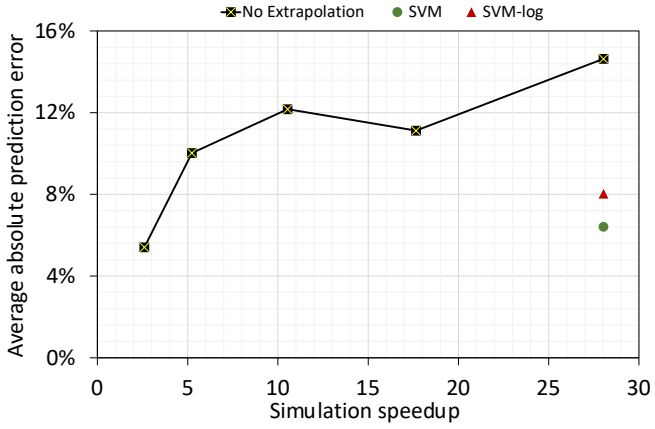


Figure 4.7: Prediction error versus simulation speedup. *SVM-based prediction and regression achieve high prediction accuracy while yielding high simulation speedups.*

quad-core, octo-core and finally a 16-core scale model. Prediction accuracy generally improves as we move towards larger scale models[2], while simulation speedup decreases considerably. The ML-based prediction techniques, SVM and SVM-log, rely on a single-core scale model simulation only, and hence yield the highest possible simulation speedup, namely 28×. Overall, the con-

[2]The dual-core scale model is more accurate than the quad-core scale model due to how memory bandwidth is scaled down, see also Table 4.1. Both the 'MC-first' and 'MB-first' scaling methods (discussed in Section 4.6.1) lead to similar trend anomalies, albeit at different core counts.

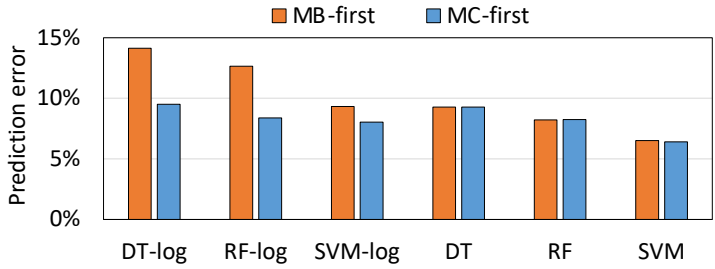


Figure 4.8: Evaluating memory bandwidth scaling alternatives under PRS. *ML-based regression achieves higher accuracy by first scaling the number of memory controllers ('MC-first') compared to first scaling memory bandwidth per memory controller ('MB-first').*

clusion is that ML-based prediction and regression is accurate while yielding high simulation speedups.

4.6 Sensitivity Analyses

We now perform a couple analyses to evaluate the sensitivity of the proposed scale-model simulation methodology. We consider the homogeneous workload mixes throughout.

4.6.1 Memory bandwidth scaling

Recall from Section 4.2 that we explored two options for how to proportionally scale down memory bandwidth from the target system to the scale model. One option ('MC-first', our default) is to first scale the number of memory controllers (while keeping memory bandwidth per memory controller constant) and then scale memory bandwidth per memory controller when there is only single memory controller left. An alternative option ('MB-first') is to first scale down memory bandwidth per memory controller from 16 to 4 GB/s while keeping the number of memory controllers constant, and then scale down the number of memory controllers from 8 to 1. Figure 4.8 reports prediction error for the various scale models under MC-first and MB-first. We find that first scaling the number of memory controllers yields the highest accuracy, especially for the ML-based regression techniques. In particular, for SVM-log, the average prediction error reduces from 9.3% to 8.0%; the improvement in accuracy is even more substantial for DT-log: reduction in average prediction error from 14.1% to 9.5%.

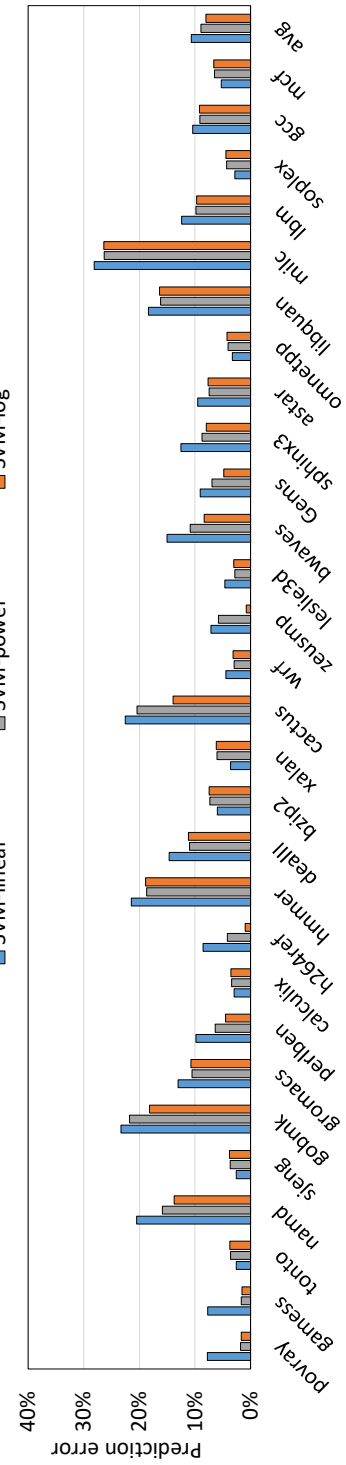


Figure 4.9: Linear, power and logarithmic regression under SVM. *Logarithmic regression yields the lowest prediction error.*

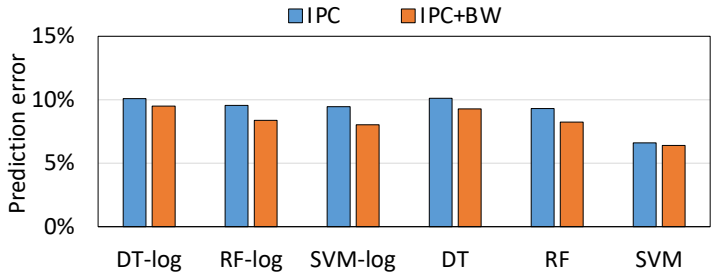


Figure 4.10: Varying the input variables to the ML-based extrapolation techniques. *Considering both performance and bandwidth utilization as input variables leads to improved accuracy compared to using only performance as input.*

4.6.2 Regression

As aforementioned in Section 4.3.2.2, we evaluated three regression approaches following a linear model ($y = a \cdot x + b$), a power model ($y = a \cdot x^b$) and a logarithmic model ($y = a \cdot ln(x) + b$), in which x is the number of cores and y is performance. We use least squares regression to obtain the parameters a and b that yield the best fitting curve. Figure 4.9 reports the accuracy for these three regression techniques under SVM-based regression. Logarithmic regression outperforms the power and linear models by a significant margin for most of the benchmarks, and leads to the lowest average prediction error: 10.7% (linear), 8.9% (power) and 8.0% (logarithmic).

4.6.3 ML model inputs

The proposed scale-model simulation methodology uses performance (IPC) and bandwidth utilization as input to the ML models, see Section 4.3. Figure 4.10 reports the average prediction error for the different ML-based prediction and regression methods when comparing using both performance and bandwidth utilization as input versus using performance only. Using both performance and bandwidth utilization improves the prediction error by a significant margin compared to using only performance. In particular, for SVM-log, the average prediction error reduces from 9.5% to 8.0%.

4.6.4 Multi-core scale-models under regression

As discussed in Section 4.3.2.2, the ML-based regression techniques use a number of multi-core scale models to drive the regression. So far, we assumed four multi-core scale models with 2, 4, 8 and 16 cores. Figure 4.11 reports the prediction error when changing the number of multi-core scale models to 2 (dual- and quad-core scale models), 3 (dual-, quad- and octo-core scale models)

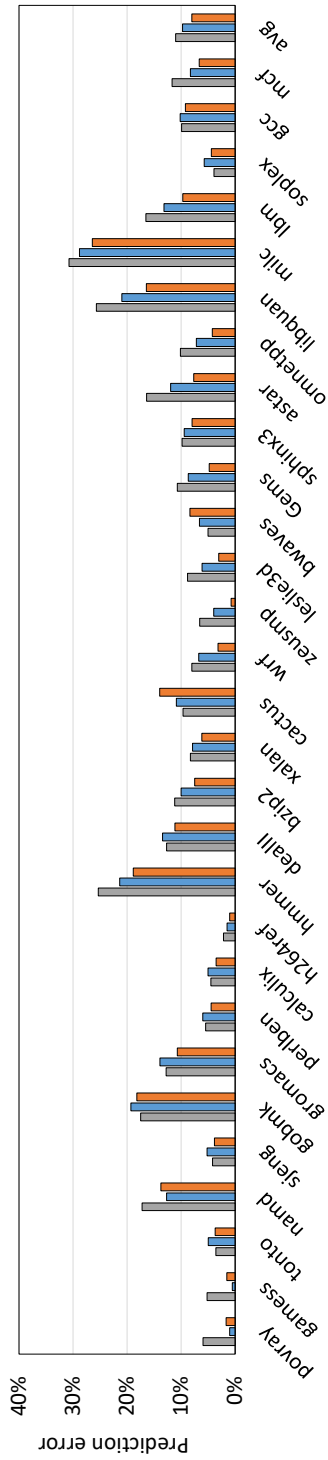


Figure 4.11: Prediction error as a function of the number of multi-core scale models used for SVM-log regression. *The prediction error only slightly increases with a reduced number of multi-core scale models.*

and 4 (our default). Reducing the number of multi-core scale models might be of interest if the goal is to reduce model training time. Remarkably, the error is only slightly higher when limiting the number of multi-core scale models. The average prediction error equals 11.0% (2 and 4-core scale models) to 9.7% (2, 4 and 8-core scale models) to 8.0% (2, 4, 8 and 16-core scale models).

4.6.5 Memory bandwidth utilization

We focused on predicting performance throughout the result section. Scale-model simulation can also be used to predict other metrics, such as bandwidth utilization. This is done by considering bandwidth utilization (rather than performance) as the dependent variable when training the ML models, see also Section 4.3. Figure 4.12 reports the prediction error for predicting memory bandwidth utilization. The result is in line with the previously reported accuracy numbers: SVM is the most accurate prediction approach (8.7% average error) and SVM-log is the most accurate regression approach (11.3% average error).

4.6.6 Multi-threaded workloads

We did not consider multi-threaded workloads in our work so far. We note though that the homogeneous workload mixes considered in this work can serve as a proxy for data-parallel multi-threaded workloads in which all threads execute the same code (on different data elements) and there is very little or no communication between threads. We hence expect that scale-model simulation would perform similarly for data-parallel multi-threaded workloads as for the homogeneous workload mixes considered here. As multi-threaded workloads incur inter-thread communication and synchronization overhead, the scale-model simulation methodology will need to be extended to take other features into account including NoC delay and congestion, coherence effects, synchronization overhead, etc. This is left as part of future work.

4.7 Related Work

The most closely related work by Eyerman et al. [59] proposes scale models for an experimental Intel processor, called PIUMA (Programmable Integrated Unified Memory Architecture), that is specifically designed for the efficient execution of graph analytics workloads. The lack of resource sharing among processor cores makes the development of scale models for this type of architecture relatively easy. More specifically, the PIUMA architecture does not have shared caches; each core has a dedicated memory controller; and a highly scalable interconnection network provides high bandwidth and low latency to

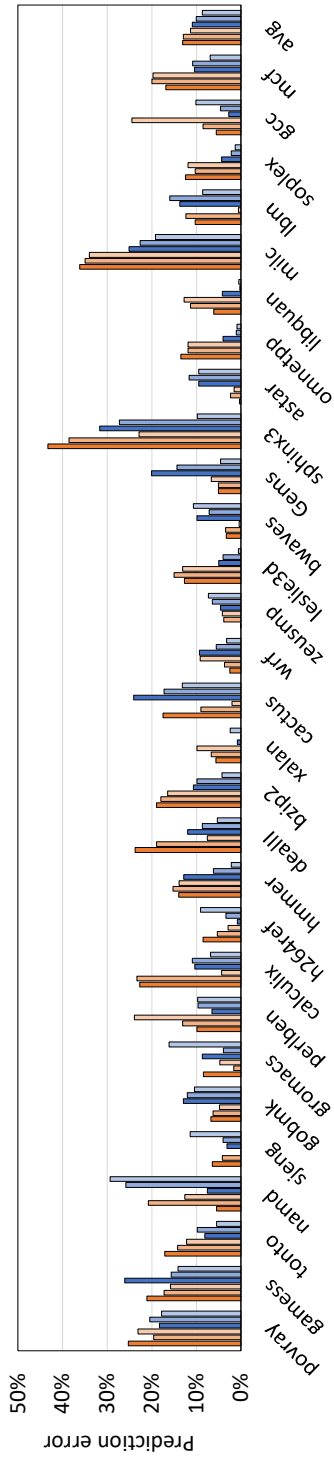


Figure 4.12: Prediction error for predicting memory bandwidth utilization. *SVM and SVM-log predict memory bandwidth utilization with an average error of 8.7% and 11.3%, respectively.*

each individual core. In contrast, the cores in a general-purpose multi-core processor share the LLC, NoC and memory subsystem.

Machine learning (e.g., neural networks [82] and spline-based regression [104]) was previously proposed to explore single-core and multi-core design spaces, however, predicting performance for larger-scale target systems fell out of reach for these models. Analytical models have been proposed for multi-core processors for both multiprogram workloads [95, 178] and multi-threaded workloads [48]. An inherent challenge for such models is how to analytically model overlap effects as well as timing-sensitive events in large target systems; scale-model simulation addresses this challenge through extrapolation. Hoste et al. [75] and Piccart et al. [144] determine the optimum platform among a set of previously benchmarked platforms for an application of interest. Other prior work predicts performance across architecture paradigms: Baldini et al. [21] and Ardalani et al. [16] propose machine-learning based methodologies to predict GPU performance based on CPU implementations.

Scaling down the workloads to speed up simulation has received considerable attention in the literature. Sampling is a widely used methodology to select representative regions of execution of an unchanged workload. Prior work has proposed sampling for single-threaded workloads [161, 188] as well as for general multi-threaded workloads [35] and barrier-synchronized workloads [37]. Alameldeen et al. [8] propose a methodology for scaling down commercial workloads in both size and runtime, allowing commodity machines to simulate much more powerful server systems. Sabu et al. [154] present a generic multi-threaded sampling methodology and achieve significant simulation speedups, which is accomplished by using loop-based markers to identify representative simulation regions and taking into account the inherent parallelism of the application. These sampling methodologies speed up simulations by scaling down the applications, which is actually complementary to the proposed scale-model simulations. However, sampling applications alone cannot solve the simulation problems of large-scale future systems.

Raising the level of abstraction is yet another, complementary, way to speed up simulation. One-IPC models assume that a single instruction is executed per cycle in the absence of miss events such as cache misses and branch mispredictions [85, 121]. Interval simulation [65] models the impact of miss events on performance through mechanistic analytical modeling. ZSim [156] and Sniper [34] implement high-abstraction simulation models for superscalar processors. While these high-abstraction models significantly speed up simulation, they do not fundamentally solve the simulation challenge of large-scale target systems.

4.8 Conclusion

This chapter proposed scale-model simulation, a novel methodology that combines architectural simulation of scale models with machine learning to predict the performance of a larger-scale target system. We provide results that demonstrate the effectiveness of scale-model simulation using both homogeneous and heterogeneous multiprogram workload mixes to predict 32-core target system performance based on single-core scale model simulation runs. We find that it is critical to proportionally scale the shared resources when constructing scale models. Leveraging ML techniques to construct extrapolation models further improves scale-model prediction accuracy. We find that ML-based regression, which does not rely on target-system simulations during training, achieves an average prediction error of 8% for homogeneous multiprogram workload mixes and 15.8% for heterogeneous mixes. Because scale-model simulation makes these predictions based on single-core scale model simulations, scale-model simulation leads to a 28× simulation speedup compared to simulating a 32-core target system using Sniper on a high-end 36-core simulation host system.

Chapter 5

Architectural Simulation of Reliability-Aware Memory Systems

The research on modern multicore processor architectures heavily relies on system simulation. Simulating modern multicore processors, however, is extremely time-consuming, especially for large core counts. Managed language workloads written in Java, Python, and C# exacerbate the simulation challenge even further and some simulations are prohibitive due to the simulation time and/or infrastructure limitations. The architectural simulation of reliability-aware memory systems in Chapter 3 also ran into this simulation problem which motivated the exploration of scale-model architectural simulation in Chapter 4.

In this chapter, we apply scale-model simulation, as proposed in Chapter 4, to predict performance for the proposed reliability-aware hybrid memory systems in Chapter 3. Simulating a 32-core system executing 32 instances of Java workloads is extremely time-consuming which takes up to one month of simulation time for several benchmarks. Moreover, we ran into simulator infrastructure issues when simulating that many cores. In fact, we were unable to complete some of the 16-core simulations and even more of the 32-core simulations. Therefore, in Chapter 3, we reported experimental results with a simplified single-core system with all shared resources scaled down proportionally. Our preliminary analysis showed that the reported results with scaled models are conservative and are supposed to be consistent with the actual results. This chapter verifies the aforementioned conclusion with a detailed architectural simulation of large-scale hybrid memory systems and presents the predicted performance using more accurate prediction models from Chapter 4. Specifically, we first construct and simulate a series of scale models of the target hybrid memory systems with reduced core count and shared resources. The

performance for the target system is then predicted using Machine Learning (ML) based regression techniques. The reason we adopt ML-based regression instead of ML-based prediction is that the training phase of ML-based prediction requires simulation runs for the target system which is impossible when simulating a 32-core hybrid memory system executing 32-instance Java workloads. The ML-based regression model, however, only requires simulation results obtained from small-scale multicore scale models, but not the target system. Applying scale-model architectural simulation to the evaluation of reliability-aware hybrid memory systems brings up new opportunities for evaluating a large-scale future computer system which is impossible to be simulated due to current simulation limitations.

This chapter is organized as follows. Section 5.1 analyzes the simulation challenges for future large-scale computer systems, especially with managed language workloads. Section 5.2 motivates the application of scale-model architectural simulations to multicore systems executing multiprogrammed Java workloads. The details of the experimental setup are then introduced in Section 5.3 which is followed by the evaluation in Section 5.4. In the evaluation section, we first provide insight into the characteristics critical to the performance increase with larger multicore simulations by generating CPI stacks for multicore system simulations with different core counts. We then evaluate the prediction accuracy of ML-based regression models to select the most accurate prediction models for the target system. The predicted performance for the target system is then presented and followed by an analysis of the evaluation of future computer systems that cannot be simulated. The conclusion of this work is discussed in Section 5.5.

5.1 Introduction

Computer architects heavily rely on detailed architectural simulation to evaluate new processor architectures. Unfortunately, architectural simulation is extremely time-consuming as even today's fastest simulators run several orders of magnitude slower than native execution. Historically, two trends have motivated advances in faster and more accurate simulation methodologies. Increasing transistor budgets enable new processors with advanced capabilities. On the software side, programming languages with higher abstractions enable new applications that solve complex societal challenges. Both trends stress architectural simulation. Modern simulation methodologies employ a variety of techniques to keep simulation times reasonable.

Researchers and practitioners employ a variety of techniques to tackle the simulation challenge, as previously discussed in this thesis. The most popular solution is sampled simulation in which a set of representative samples are simulated in detail [161, 188]. Other techniques include high-abstraction simulation [34, 156], parallel simulation [121, 123], analytical modeling [36, 156], statistical simulation [53], among others. Although successfully applied for

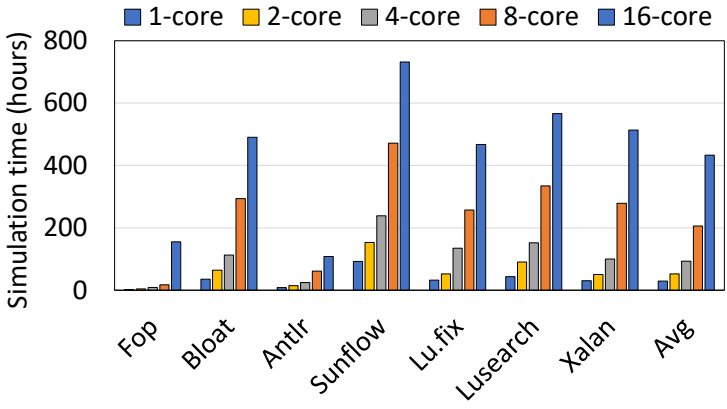


Figure 5.1: Simulation time in hours for multiprogram Java workloads with up to 16 cores on Sniper. *Simulation time of multiprogram workloads is prohibitive and increases super-linearly with an increasing number of cores.*

native language workloads, such as C and C++, these techniques are not as easily deployed for managed language workloads written in Java, Python or JavaScript. The dynamic nature and the tight interaction between application code and the underlying runtime complicate the task of identifying a short-running, yet representative workload for simulation. Current 'good practice' advice is to simulate the entire workload [30], which leads to prohibitive simulation times.

We find that simulating a single 16-core Java workload using Sniper [36], a fast and state-of-the-art parallel multicore simulator, takes more than two weeks on average, see Figure 5.1 which reports simulation time in hours of real time for a set of multiprogram DaCapo Java workloads[29]. (We provide details about our experimental setup later in Section 5.3.) We consider multiprogram workloads in this work to mimic today's cloud infrastructures which commonly run multiple independent services on a server, each within their own language runtime [186]. As Figure 5.1 reports, simulation time increases super-linearly with the number of application instances in the workload mix, and quickly becomes impractical for modern-day multicore systems.

Scale models are scaled-down but functionally equivalent versions of the (much) larger target multicore system. Scale models are chosen such that simulating a scale model takes substantially less time than the target system. More specifically, a scale model of a target multicore processor features a reduced number of cores, say by a factor N. The shared resources, in particular the last-level cache (LLC), interconnected network (NoC) and memory bandwidth, are also reduced proportionally by a factor N. Next, we simulate the scale model in detail. Assuming no resource contention in the LLC and memory sub-system, the performance of a single core in the scale model would be identical to the performance of an individual core in the target system. Of course, in reality, the actual performance will be less because of resource contention in the

shared resources, i.e., due to limited cache capacity, conflict misses, memory bandwidth, etc. Hence, naively assuming that resource contention does not affect per-core performance leads to inaccurate scale-model simulation. We thus need more sophisticated extrapolation techniques.

We leverage Machine Learning (ML), regression in particular, to extrapolate and predict larger-scale system performance based on the scale model(s), as discussed in Chapter 4. Recall that the ML-based regression involves three phases called training, prediction and regression. We evaluate a variety of ML techniques including decision tree, random forest and Support Vector Machines (SVM) models in the prediction phase and we explore linear, power and logarithmic models in the regression phase. We find that SVM with logarithmic regression (SVM-log) performs best among the ML-based regression techniques, thus we select SVM-log model to predict the performance for the target 32-core hybrid memory systems. We conclude that the reported performance for the proposed hybrid memory systems obtained from a scaled single-core simulation (see Chapter 3) is conservative, that is, the actual improvement in performance through RiskRelief compared to the baseline DRAM-Only system is higher than what single-core scale model simulations suggest. The experimental results show that RiskRelief-Nursery (RR-N) improves performance of the hybrid HBM-DRAM memory system by 62% rather than 20% over the DRAM-Only memory system. RiskRelief-Mature (RR-M) improves the performance by 68% instead of 29% compared to the DRAM-Only system.

We make the following contributions in this work:

- We apply scale-model architectural simulation that relies on scaled-down variants of the target multicore system to the evaluation of reliability-aware hybrid memory systems, which takes much less simulation time.
- We evaluate scale-model architectural simulation for emerging multiprogrammed Java workloads and demonstrate the high prediction accuracy of our proposed approach.
- Scale models enable us to use them to analyze the scalability behavior of multiprogrammed Java workloads. We bring new insights into the scaling behavior of multiprogrammed managed language workloads with increasing number of cores.

5.2 Motivation and Opportunity

5.2.1 Multicore Simulation

The de facto standard in computer architecture research and development is to pursue detailed cycle-level simulation. Unfortunately, detailed cycle-level simulation does not scale with system complexity (i.e., increasing core counts).

Raising the level of abstraction and parallel simulation are employed to speed up multicore simulation, as exemplified by Sniper [36] and ZSim [156]. In spite of these enhancements, simulation time is still problematic, especially when simulating large multicore systems and running long-running workloads (e.g., managed language workloads), as reported in Figure 5.1. Note that simulation speed is not limited by the number of cores and/or the available memory capacity in the simulation host: Sniper is a parallel simulator that significantly benefits from the available number of cores in our 32-core simulation host.

More specifically, Sniper [36] uses the mechanistic interval-analysis core model [65] to improve simulation speed by an order of magnitude. Reduced simulation time with mechanistic models allows architects to explore and propose optimizations for managed languages that use a higher level of abstraction than native C and C++ applications [2, 3, 4, 5, 7, 41, 151, 158, 163].

Sniper is a parallel simulator, which enables faster simulation of multiprogrammed workloads. Internally, the simulator maintains per-core data structures, and each core advances its instruction stream independently of the other cores. Periodically, the cores synchronize (barrier-synchronization interval) to correctly advance global (simulated) time, and to synchronize access to shared resources such as the last-level cache. Although the parallel nature of Sniper allows fast simulation, as Figure 5.1 shows, synchronization inhibits perfect scalability of simulation times for multiprogrammed Java workloads. Note further that the simulator's performance and scalability are also inhibited by limitations in the host environment, such as the number of cores and the physical memory.

5.2.2 Java Workload Simulation

Java exposes a higher level of abstraction to the programmer than the C language. To facilitate faster development times and to provide platform independence, a runtime environment, i.e., the Java Virtual Machine (JVM) provides bytecode interpretation, just-in-time compilation, memory safety, and security. These services run in their own context, and increase the overall execution time of a Java program. For instance, garbage collection can cost up to 30% of the total execution time [2, 33, 132]. Therefore, simulation times of Java programs are high, partly due to the presence of a language runtime. Higher abstraction object-oriented languages encourage copious object allocation [5, 194], which stresses the memory sub-system, and consequently increases simulation times. The single-programmed Java workloads we simulate execute up to 30 billion instructions. Multiprogramming increases the simulation workload due to contention for shared resources, which results in even higher simulation times.

As can be deduced from Figure 5.1, simulation times increase tremendously beyond 4-core scale models. These high simulation times are partly because the state-of-the-art Java simulation methodology is execution-driven, and the methodology is to execute the entire Java application. Unlike C benchmarks,

no prior work has experimented with sampling-based approaches for Java applications, which further motivates our scale-model architectural simulation for multiprogrammed Java workloads.

5.2.3 CPI Stacks

Ideally with a core-count-proportional multicore architecture, the increase in total execution time (i.e., the last application to finish) when running multiple instances of the same application should be negligible. However, contention for shared resources such as the last-level cache, and the shared memory bandwidth change the picture. To visualize the performance scaling bottlenecks in multiprogramming workloads, we use cycles per instruction (CPI) stacks [57]. CPI stacks divide the total CPI into components. Each component represents the impact of a certain event on total performance. The bottom component is the base component which represents the instruction-level parallelism inherent in the simulated program. The other components show the impact of stall events such as cache misses at different levels of the hierarchy. The length of a component is proportional to its impact on overall performance. All components are stacked in the form of a stack with the base component typically shown at the bottom. We use CPI stacks to analyze the scaling behavior of multiprogrammed Java workloads, because they are an intuitive way to visualize performance scaling bottlenecks and provide insights to the parameters measured in the performance prediction.

5.3 Experimental Setup

5.3.1 Simulator and Java Virtual Machine

We use Sniper v6.0, a parallel and high-speed cycle-level x86 simulator for multicore systems, using its most detailed cycle-level hardware-validated core model [36]. Prior work extended Sniper for managed language runtimes, including dynamic compilation and emulation of frequently-used system calls [158].

We use Jikes RVM 3.1.2 and best practices from prior work to evaluate Java workloads [30]. We focus on steady-state performance and therefore use replay compilation to eliminate non-determinism due to the VM's dynamic optimizing compiler. See Chapter 3 for more specifics.

5.3.2 Simulated Processor Architectures

The processor architecture that we evaluate in this chapter is consistent with that in Chapter 3. Specifically, we simulate processors with up to 32 cores in this work, see Table 5.1 for the processor's key architecture parameters. We simulate out-of-order cores with a 3-level cache hierarchy. We evaluate three

Processor	
Number of cores	32 cores
Core frequency	4.0 GHz
Issue width	4-wide
ROB size	128 entries
Branch predictor	hybrid local/global predictor
Max. outstanding	48 loads, 32 stores, 10 L1-D misses
Cache Hierarchy	
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
LLC	shared 32 MB, 64 way, 30 cycle NUCA, 32 slices, 1MB/slice, 1 slice/core
NoC	
Mesh topology	4×8
Bandwidth	128 GB/s bisection bandwidth
HBM	
Capacity	2 GB for hybrid, 32 GB for HBM-only
Bandwidth	128 GB/s aggregate bandwidth
ECC	SEC-DED ECC [76]
tCAS-tRCD-tRP-tRAS	45-45-45-180 CPU cycles
DRAM	
Capacity	32 GB
Bandwidth	25.6 GB/s aggregate bandwidth
ECC	single-ChipKill ECC [49]
tCAS-tRCD-tRP-tRAS	45-45-45-180 CPU cycles

Table 5.1: Target system parameters.

memory systems: DRAM-Only, HBM-Only (both with 32 GB of main memory) and a hybrid HBM-DRAM system with 2 GB HBM and 32 GB DRAM, see also Table 5.1. We consider two proposed garbage collectors to manage the hybrid memory systems: RiskRelief-Nursery (RR-N) and RiskRelief-Mature (RR-M), which can be recalled in Chapter 3. We further assume a shared 32 MB NUCA cache, a 128GB/s bisection bandwidth mesh NoC, 25.6 GB/s DRAM bandwidth and 128 GB/s HBM bandwidth. We assume SEC-DED ECC for HBM because of its lower complexity and power consumption; we use single-ChipKill ECC for DRAM to be in line with production systems. As previously discussed, we scale the number of cores, the size of the LLC and memory bandwidth proportionally as we construct scale models of the target multicore processor.

5.3.3 Workloads

We use the same Java workloads for the evaluation in this chapter with that in Chapter 3. Specifically, we use 9 applications from the DaCapo suite [29] that work with our simulation and VM infrastructure. We use four benchmarks

from the DaCapo-9.12-bach benchmark suite (sunflow, lusearch, pmd and xalan). We use an updated version of lusearch, called lu.Fix [189], that eliminates useless allocation, and an updated version of pmd, called pmd.S [52], that eliminates a scaling bottleneck due to a large input file. We use three benchmarks from DaCapo 2006: fop, antlr and bloat. As in established methodology, we use 2× the minimum heap size for our benchmarks, and we use the default inputs for measurements.

In addition to the workloads, the evaluation methodology for Java workloads is also consistent with that in Chapter 3. We create multiprogram workloads with up to 32 application instances. Each workload runs multiple instances of the same application, i.e., a so-called rate workload. We synchronize the start of execution of the different instances in the workload using a barrier. We do not restart instances after they finish execution. We use the average execution time of the various instances as a measure for per-application performance. Prediction error is computed as the absolute relative difference between the predicted execution time versus the measured execution time.

5.3.4 Scale-Model Simulation

Scale-model architectural simulation involves two key phases: scale-model construction and scale-model extrapolation. For scale-model construction, we keep consistent with the methodology introduced in Chapter 4. For scale-model extrapolation, we adopt the Machine Learning (ML) based regression model. The ML-based prediction model requires simulation results for the target system during the training phase while it is prohibitive to simulate 32-core performance for the Java workloads on the proposed hybrid memory system due to infrastructure limitations. The ML-based regression model, however, does not have requirements to the simulations for the target system and the simulation results of the target system can be predicted by the regression models.

To select the most accurate ML-based regression models to predict performance for the target 32-core memory system, we need to first predict performance for a small-scale target system to compare the prediction accuracy among all evaluated models. We evaluate the prediction accuracy for an 8-core scaled system to facilitate the selection of prediction models. The reason of evaluating 8-core systems rather than 16-core systems is that there are simulation limitations on the 16-core system for pmd and pmd.S. We also miss the 16-core simulation results for pmd and pmd.S in the figures shown in Section 5.4 for the same reason.

5.4 Evaluation

We now evaluate scale-model simulation for multiprogrammed managed language workloads. We first focus on the CPI stacks to explore the reasons

for the increase in execution time with increasing core counts, which can help to provide insights into the selection of measured characteristics for the prediction model. Then we evaluate the prediction accuracy of the ML-based regression model for a small-scale (8-core) target system, which can help to select the most accurate prediction models to predict performance for a larger target (32-core) system. Finally, we predict the performance for the target system and compare the predicted performance with the simulation results in Chapter 3.

5.4.1 CPI Stacks

Accurate estimation requires identifying the reasons for the increase in execution time. As shown in Figure 5.2, we identify three components in the CPI stack that are the reasons for the increase in execution time: (1) increase in instruction fetch latency, (2) increase in execution time due to contention in the last-level cache, and (3) increase in DRAM access latency. Since the core microarchitecture of the scale model is similar to the target multicore, there is no change in the base component, and components that pertain a full issue queue etc. Next, we discuss in detail the three components of the CPI stack, which are responsible for the increase in execution time of a multiprogrammed Java workload.

Java applications demand high instruction fetch throughput [129]. This is because object-oriented program design encourages modularity, and thus results in programs with a large number of methods. Jumping between methods leads to high instruction cache miss rates. Multiprogramming increases the probability of off-chip DRAM accesses to fetch instructions. Since DRAM latency is an order of magnitude longer than the access latency to the last-level cache, the instruction fetch component of the CPI stack increases as we add more instances to the Java workload.

The scale model methodology proportionally increases the shared last-level cache size. Nevertheless, sharing in the last-level cache leads to contention, which means more cache lines need to be fetched from DRAM, increasing the overall execution time. For our chosen Java applications, the contribution of the L3 cache latency to the CPI is not significant. However, for several applications such as Sunflow and Lu.fix, we still observe an increase in the L3 cache latency.

The third component that changes with more cores is DRAM access latency. We model DRAM access latency as the sum of: (1) the data transfer latency which depends on the transfer medium, (2) the access time which depends on whether the memory request is resolved in the row buffer or the memory array, and (3) the queuing delay which depends on memory bandwidth. Memory bandwidth is dictated by pin count, power and packaging constraints. Therefore, if the cores generate more requests than the memory system can sustain, the requests queue in the memory controller. The data transfer latency in our model is fixed; the access time is distinguished by row buffer hit or miss; whereas the queuing delay varies on a per-request basis. Generally, the queuing

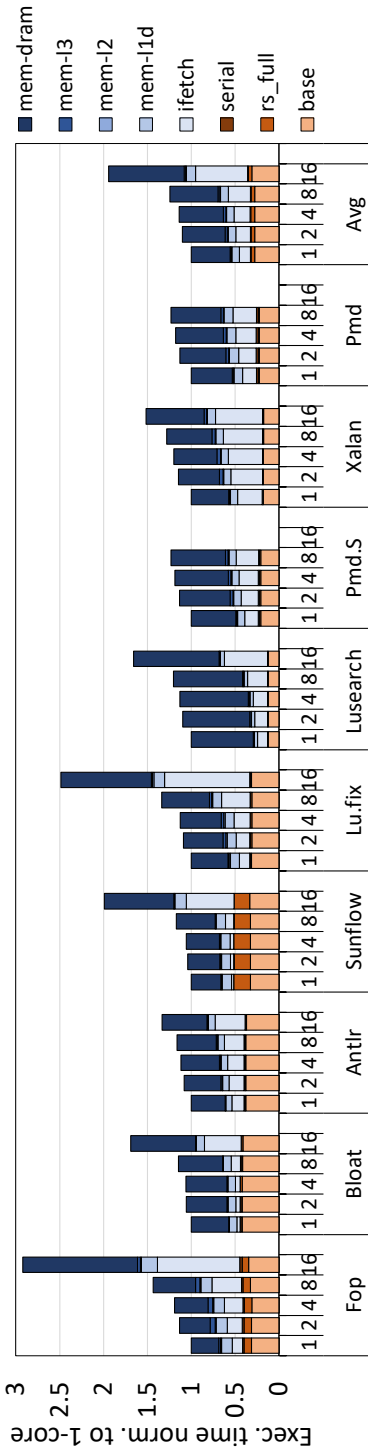


Figure 5.2: The CPI stacks for multi-core systems normalized to a single-core system. *The instruction fetch latency and DRAM access latency have a large contribution to CPI and keep increasing with system scaling. The access latency to last-level cache (LLC) also has a significant increase with increased core counts but it only takes 1% of the total execution time on average.*

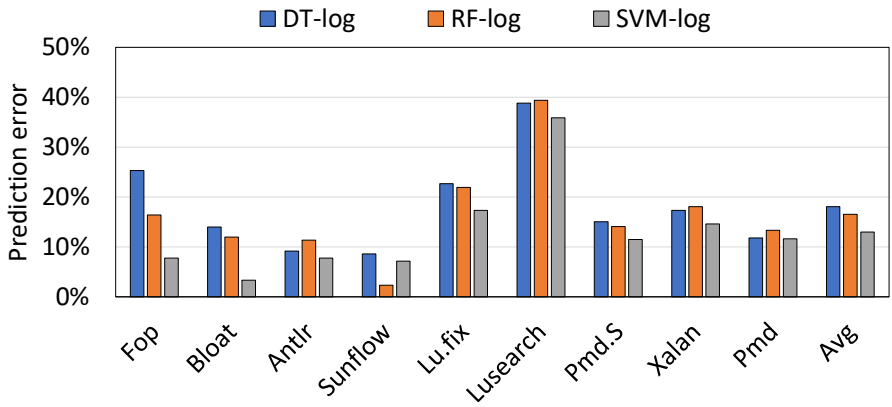


Figure 5.3: Prediction error for the 8-core system using ML-based regression models. *SVM with logarithmic regression (SVM-log) yields the highest prediction accuracy with an average prediction error of 13.0% and at most 35.8%.*

delay increases by 2× on average from a 1-core scale model to a 4-core scale model of a 32-core multicore processor.

5.4.2 Model Selection

Recall that we proposed two ML-based extrapolation models in Chapter 4: the ML-based prediction and the ML-based regression. The training phase of ML-based prediction requires simulation results for the target system. However, running multiprogrammed Java workloads on a 32-core target system is extremely time-consuming and impossible for some benchmarks due to the simulator infrastructure issues. On the other hand, ML-based regression needs no simulation runs of the target system and the performance of target system can be predicted by the regression phase. Therefore, ML-based regression is suitable for predicting the performance of a 32-core system that executes multiprogrammed Java workloads.

We consider three machine learning models: Decision Tree (DT), Random Forest (RF) and Support Vector Machines (SVM) during the prediction phase and we use logarithmic regression for the regression phase. Considering that the loss of 32-core simulation results makes it impossible to compare the prediction accuracy of the aforementioned techniques, we need prediction accuracy for a small-core system to determine which model is the most accurate to predict performance for the 32-core target system.

Figure 5.3 reports the prediction error for an 8-core system using the aforementioned ML-based regression techniques. SVM is the most accurate ML-based regression model with an average error of 13.0% and a maximum error

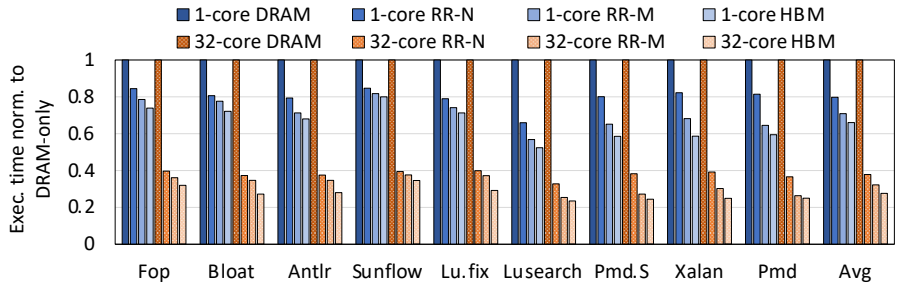


Figure 5.4: Scaled and predicted 32-core execution time normalized to DRAM-Only for the RiskRelief collectors and HBM-Only. *The 1-core performance results are obtained from a single-core system with all shared resources scaled down proportionally. The 32-core performance results are predicted using small-core simulation results and the SVM-log regression model.*

of 35.8% for Lusearch. RF produces an absolute prediction error of 16.5% on average and up to 39.4%. DT is the least accurate model of the three models with an average error of 18.0% and a maximum error of 38.8%. The general conclusion is consistent with what we obtained in Chapter 4 — namely, SVM with logarithmic regression (SVM-log) yields the highest accuracy among the ML-based regression techniques. Unless otherwise stated, we report the predicted performance for the 32-core target systems using the SVM-log technique in the remaining parts of this chapter.

5.4.3 Performance

We report the performance results for the scaled single-core system and the target 32-core system in Figure 5.4, normalized to their corresponding DRAM-Only systems. The 1-core performance results are collected from the simulation of single-core scale models without extrapolation, where the shared resources, the last-level cache (LLC), interconnection network and memory bandwidth in particular, are reduced proportionally. These numbers are also consistent with those shown in Figure 3.10 in Chapter 3 and we use them as a comparison with the following 32-core performance results. The 32-core performance results are firstly predicted using the SVM-log regression model, and then the performance numbers for the RiskRelief collectors and the HBM-Only system are normalized to the DRAM-Only system.

Given a 32-core target system with an HBM-Only memory, execution time collected from a single-core scale model reduces by 34% and up to 48% (Lusearch) compared to a DRAM-only memory. As a comparison, execution time predicted for the target system – predicted using single-core simulation results and ML-based regression techniques – is reduced by a significant margin com-

Figure 5.5: Execution times for the 32-core target system collected through scaled-down simulations without extrapolation (the first 5 sets of bars) and predicted using Machine Learning based regression techniques (the last 2 sets of bars): RiskRelief-Nursery, RiskRelief-Mature and HBM-Only normalized to DRAM-Only. *The performance benefits from RiskRelief collectors and HBM-Only increase with larger core simulations over DRAM-Only, and the predicted target performance confirms this performance benefit tendency.*

pared to a DRAM-only memory, with an average of 72% and at most 77%. The reason for a larger performance gap with ML-based regression models compared to the single-core system is that the single-core simulations assume shared resource contention observed in the single-core model is the same as in the target system. The contention, in reality, is supposed to be more intense and therefore puts more pressure to the memory system than assumed. Consequently, the performance benefits obtained from HBM are expected to be much more than we observed in Chapter 3. In other words, HBM-only can improve performance by 72% instead of 34% compared to DRAM-only.

For the RiskRelief-Nursery (RR-N) collector, the average execution time reduces by 20% for a single-core system and by 62% for the 32-core target system compared to a DRAM-Only system. To further benefit from the HBM bandwidth, RiskRelief-Mature (RR-M) reduces the execution time on average by an additional 9% for a single-core system and 6% for the 32-core system over RR-N by partitioning mature space and large object space into DRAM and HBM regions. The percentage numbers are 29% and 68% on average compared to a DRAM-Only system, respectively. Across all results reported in Figure 5.4, we confirm that the performance tendency of a 32-core target system is consistent with what we concluded in Chapter 3 — performance results for a scaled single-core system are conservative and the actual performance benefits from RiskRelief collectors for a 32-core target system are higher than what the single-core simulations in Chapter 3 suggested.

5.4.4 Large Target System Prediction

For target systems that cannot be simulated (the reliability-aware 32-core memory systems in our case), the evaluation statistics like performance can be either collected by small-core scale-model architectural simulation with no extrapolation, or predicted by Machine Learning (ML) based regression techniques.

Figure 5.5 reports the execution time for four 32-core target systems (i.e., DRAM-Only, HBM-Only, RiskRelief-Nursery and RiskRelief-Mature) using the aforementioned methodologies normalized to a DRAM-Only system. Small-core simulations with no extrapolation are constructed with a small-scale core count ranging from a single core up to 16 cores with shared resources scaled down proportionally. The proposed reliability-aware techniques, RR-N and RR-M, yield higher performance benefits compared to a DRAM-Only system at higher core counts because resource contention is better simulated and then benefits more from HBM bandwidth with more cores. Grouped bars on the right for each benchmark in Figure 5.5 report the execution time for 16-core and 32-core systems predicted using ML-based regression models — SVM-log, specifically. The predicted 16-core performance results for the RiskRelief collectors and the HBM-Only system compared to the DRAM-Only system represent conservative benefits compared to the simulation performance for the

16-core simulations. The 32-core predictions demonstrate higher performance improvements than the 16-core predictions.

All together, small-core scale-model simulation is an effective alternative option to a large-scale system where the simulation is time-consuming or even impossible to launch. The performance tendency of scale models is consistent with the real target system but with some conservative numbers. ML-based regression techniques can improve the prediction accuracy of performance for the target system and bring the predicted performance results closer to the simulated results obtained from an actual target system. Scale-model architectural simulation with Machine Learning based regression techniques opens up a new avenue to explore and evaluate promising techniques on a future large-scale system where the simulation is time-consuming or technically prohibitive.

5.5 Conclusion

In this chapter, we apply scale-model simulation, a novel methodology that combines architectural simulation of scale models with machine learning based extrapolation techniques, to evaluate the proposed reliability-aware garbage collection implemented on a large-scale hybrid memory system with multiprogrammed managed language workloads executing on it. The experimental results present the effectiveness of ML-based regression techniques in predicting the performance of a 32-core target hybrid memory system that is impossible to simulate due to limitations of the simulator infrastructure. We verify that previous simulation results collected from single-core models reported in Chapter 3 are indeed representative and conservative results for the real 32-core target system. The improvement in performance benefiting from RiskRelief garbage collectors is suggested to be higher than what is reported in Chapter 3. We find that placing the nursery space in the HBM partition (RR-N) yields a performance benefit of 20% on average based on the single-core simulations and the improvement number is 62% for the 32-core target system using SVM-based logarithmic regression compared to DRAM-only. RR-M further improves average performance by 9% over RR-N and by 29% over DRAM-only shown from the single-core results and it achieves at 6% over RR-N and 68% over DRAM-only for the target system.

Chapter 6

Conclusion and Future Work

We pass through the present with our eyes blindfolded. We are permitted merely to sense and guess at what we are actually experiencing. Only later when the cloth is untied can we glance at the past and find out what we have experienced and what meaning it has.

– Milan Kundera

This chapter concludes the key contributions drawn from this dissertation and provides several potential avenues for future work.

6.1 Summary

Conventional DRAM memory is experiencing severe scaling challenges derived from emerging applications and memory trends. On the application side, emerging workloads deliver increasing requirements for memory bandwidth and capacity. Big data analytics, for example, use advanced analytic techniques to uncover information from large data sets, requiring efficient manipulation of large amounts of data. On the main memory side, manufacturing complexity has encouraged hybrid memories that combine HBM and DRAM to deliver high bandwidth and large capacity. However, the soft error reliability, especially for HBM, is becoming a concern without proper management while it attracts limited attention. In this thesis, we propose reliability-aware memory management for hybrid HBM-DRAM memories to minimize the soft error rate

while maximizing the overall performance. The key design is to predict hot and low-risk objects which are then placed in the HBM space to improve system reliability and performance.

System simulation is an enduring problem and computer architects have made unremitting efforts to improve prediction accuracy and reduce time overhead. Unexpectedly but reasonably, simulation challenges hit our research on reliability-aware memory management when we evaluate the proposed techniques on the hybrid memory. Specifically, simulation is excessively time-consuming for a large-scale system, and the infrastructure limitation makes things worse – it is even impossible to simulate the target system with some unsolvable constraints such as insufficient computing and memory capacity. We solve the aforementioned problems by introducing scale models into the architectural simulation of large-scale future systems. Scale-model simulation combines architectural simulation with machine learning techniques to predict performance for large-scale systems, achieving high prediction accuracy with limited simulation time overhead.

Allocation-Site Prediction for Object Hotness and Risk. Exploring the performance and reliability trade-offs for the memory system depends on the profiling of memory data. The proposed hybrid HBM-DRAM memory system is automatically managed by garbage collection at an object-level granularity. We propose the notion of hotness and risk as the performance and reliability metrics of an object. The profiling reveals a relatively weak correlation between object hotness and risk, thus both metrics need to be considered for object classification and prediction. We further propose a profiling framework to measure object hotness and risk on a per allocation-site basis. It turns out that hotness and risk present high homogeneity across objects allocated from the same allocation site. Therefore, we conclude that allocation site is an accurate predictor to predict whether objects are hot and low-risk for placement in the HBM memory.

Reliability-Aware Garbage Collection for Hybrid Memories. Garbage collection relieves the programmer from the burden of manually managing memory. In this thesis, we explore garbage collection to balance reliability and performance for a hybrid HBM-DRAM memory system. We propose two reliability-aware garbage collectors to predict and allocate hot and low-risk objects in HBM. Both RiskRelief-Nursery (RR-N) and RiskRelief-Mature (RR-M) place the nursery space for young objects in HBM because the nursery is highly accessed and low-risk. RR-M further places hot and low-risk mature objects in HBM using allocation-site prediction. RR-N achieves an averaged 20% performance benefit compared to a DRAM-only system through placing nursery objects in HBM, and the overall soft error rate is reduced by 18x compared to a HBM-only system through keeping the remaining objects in DRAM. RR-M obtains an additional 9% performance benefit over RR-N and reduces SER by 9x over HBM-only with the hot and low-risk mature objects also placed in HBM. Both RR-N and RR-M eliminate page migration overheads, substan-

tially improving performance compared to the state-of-the-art OS approach for reliability-aware data placement.

Scale-Model Architectural Simulation for Native Language Workloads. We propose scale-model architectural simulation to predict performance for the larger-scale future system. One key insight behind this proposal is that it is critical to proportionally scale the shared resources when constructing scale models. Leveraging machine learning techniques to construct extrapolation models further improves the prediction accuracy of scale-model simulation. We construct two ML-based extrapolations called ML-based regression and ML-based prediction, respectively. The evaluation on SPEC CPU workloads presents that scale-model simulation leads to a 28× simulation speedup compared to simulating a 32-core target system. In addition to the substantial simulation speedup, scale-model simulation is also effective and accurate. ML-based regression achieves an average prediction error of 8% for homogeneous multiprogrammed workload mixes and 15.8% for heterogeneous workload mixes. ML-based prediction is slightly more accurate compared to ML-based regression, with an average prediction error of 6.4%, as it involves target-system simulations during the training phase.

Scale-Model Simulation for Managed Language Workloads. The initial motivation of scale-model simulation is the simulation challenges that we encountered when evaluating the proposed garbage collection on a hybrid memory system. More specifically, it is very time-consuming and often prohibitive for multiprogrammed managed language workloads to be executed on a 32-core hybrid memory system due to simulator infrastructure constraints. We tackle this simulation problem by constructing scale-model simulation for the target hybrid memory to relieve host limitations and reduce simulation time overhead. The experimental results illustrate the feasibility of predicting performance for a designed system that cannot be simulated. Moreover, we conclude that the evaluation results obtained from a proportional scale model without ML-based regression were conservative. When extrapolating to the 32-core system, a hybrid memory system with RR-N and RR-M memory management are expected to improve performance by 62% (instead of 20%) and 68% (instead of 29%) over conventional memory, respectively.

6.2 Future Work

In this section, we discuss several promising directions for future work, mainly focusing on the improvements of reliability-aware garbage collection, the application of reliability-related optimizations to a heterogeneous system, and the extensions for scale-model simulation.

Reliability-Aware Garbage Collection. We propose two reliability-aware garbage collectors that exploit allocation-site prediction to place hot and low-risk objects in HBM and the rest in DRAM. These garbage collectors rely

on an offline profiling framework to measure object hotness and risk on a per allocation-site basis. Offline profiling efficiently guides data placement while unseen allocation site might exist during the profiling. Specifically, an allocation site may not be executed in the profile run while it is executed in a production run. Our current solution is to place objects allocated from unprofiled allocation sites in DRAM by default to minimize the soft error rate. Future work could be conducted from two aspects: (1) exploring the distribution of unprofiled allocation sites; and/or (2) dynamically profiling allocation sites. The former direction focuses on the homogeneity analysis of unprofiled allocation sites. There would be a performance loss by simply placing objects from unprofiled allocation sites in DRAM if these sites were proven to be heterogeneous, especially when a considerable number of them are hot and low-risk. For the second potential, it could be feasible by dividing the execution of the application into a sampling phase and a normal execution. The garbage collector monitors reads and writes to the objects during the sampling phase and the sampling information is stored in the object headers by the compiler. During the normal execution, the collector places hot and low-risk objects in HBM and the rest in DRAM according to the sampling information.

Profiling work requires one to record the number of reads and writes for each object. We adopt dynamic binary instrumentation to profile per-object access frequency using Pin [113]. The drawback of Pin is that it does not have a notion of an object's boundary in memory. Therefore, we need to aggregate memory access logs from Jikes RVM and Pin, then create the access trace which contains all objects allocated from a certain allocation site and the total number of accesses to each object on a per allocation-site basis. One promising alternative is to employ read and write barriers in the managed runtime. Generational garbage collectors use reference write barriers for correctness [177, 190]. Write barriers record all mature-to-nursery pointers in a remembered set, which are processed during a minor collection to precisely identify all live nursery survivors. Primitive write barriers are a straightforward extension of reference write barriers. Prior work shows that the overhead of write barriers is low, ranging from less than 1% to 3% on modern hardware [190]. On the other hand, read barriers are limited to be used in existing garbage collectors due to their huge overhead [102]. Jikes RVM, the Java virtual machine that we used in this thesis, provides both primitive and reference write barriers [15] while lacking the implementation of read barriers.

Reliability Improvements for Heterogeneous Systems. Heterogeneous multicores [101, 102] are widely explored and optimized these days for different targets including high performance, low power consumption, etc. Prevalent designs involve multiple core types such as high-performance (big) cores and energy-efficient (small) cores to allow flexibility in the power-performance trade-off of a processor. Other work, on the other hand, pays attention to the trade-off between performance and reliability for a heterogeneous system. For example, prior work [131] observed that applications exhibit different soft error reliability characteristics on big cores and small cores, and proposed a

reliability-aware scheduler to dynamically schedule applications on different cores to improve the overall system reliability.

In this thesis we propose reliability-aware garbage collection to improve performance and reliability on heterogeneous memories. One avenue for future work is to employ reliability-aware garbage collectors in a system where both processors and memory are heterogeneous. We could propose a novel reliability-aware scheduling policy to improve the overall system reliability while achieving high performance. On the application side, we evaluate reliability-aware garbage collection using Java applications from the DaCapo benchmark suite. Emerging big-data platforms such as Apache Spark are also implemented by managed programming languages, which can be further scheduled and evaluated on a heterogeneous system using the proposed methodology.

Scale-Model Simulation. We proposed scale-model architectural simulation to predict performance for future large-scale systems. The evaluations are performed using both homogeneous and heterogeneous multiprogrammed workload mixes generated from SPEC CPU workloads. We also simulate a large-scale system with hybrid memory using multiprogrammed Java workloads. This novel simulation methodology provides a range of opportunities for future work. On the application side, we could extend scale-model simulation for multithreaded applications. Challenges focus on quantifying the effects of more interference factors, such as inter-thread communication and synchronization overhead, on the simulation model construction. On the architecture side, future work could extend scale-model simulation to other architectures including throughput processors such as GPUs. On the commercial field, scale-model simulation could help to make procurement and purchasing decisions as it is feasible for scale-model simulation to predict performance for next-generation processors and systems.

Bibliography

[1] S. Akram, J. B. Sartor, and L. Eeckhout. DVFS performance prediction for managed multithreaded applications. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 12–23, 2016.

[2] 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)*, 13(1):1–25, 2016.

[3] S. Akram, J. B. Sartor, and L. Eeckhout. DEP+BURST: Online DVFS performance prediction for energy-efficient managed language execution. *IEEE Transactions on Computers (TC)*, 66(4):601–615, 2017.

[4] S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout. Managing hybrid memories by predicting object write intensity. In *Proceedings of the International Conference on the Art, Science, and Engineering of Programming (Programming'18)*, pages 75–80, 2018.

[5] S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout. Write-rationing garbage collection for hybrid memories. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 62–77, 2018.

[6] S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout. Emulating and evaluating hybrid memory for managed languages on NUMA hardware. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 93–105, 2019.

[7] S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout. Crystal Gazer: Profile-driven write-rationing garbage collection for hybrid memories. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS)*, 3(1):1–27, 2019.

[8] A. R. Alameldeen, M. M. Martin, C. J. Mauer, K. E. Moore, M. Xu, M. D. Hill, D. A. Wood, and D. J. Sorin. Simulating a $2 M commercial server on a $2 K PC. *Computer*, 36(2):50–57, 2003.

[9] D. Alexandrescu, E. Costenaro, and M. Nicolaidis. A practical approach to single event transients analysis for highly complex designs. In *Proceedings of IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 155–163, 2011.

[10] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1): 211–238, 2000.

[11] B. Alpern, S. Augart, S. M. Blackburn, M. A. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. J. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. A. Ngo, V. Sarkar, and M. Trapp. The Jikes RVM Project: Building an open source research community. *IBM System Journal*, 44(2):399–418, 2005.

[12] Amazon Web Services, Inc. AWS Lambda. `https://aws.amazon.com/lambda/`.

[13] AMD. High bandwidth memory. `https://www.amd.com/en/technologies/hbm`.

[14] AMD. BIOS and kernel developers guide for AMD NPT family 0Fh processors. Technical report, AMD, 2007. URL `https://www.amd.com/system/files/TechDocs/32559.pdf`.

[15] A. W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and experience*, 19(2):171–183, 1989.

[16] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu. Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pages 725–737, 2015.

[17] E. K. Ardestani and J. Renau. ESESC: A fast multicore simulator using time-based sampling. In *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 448–459, 2013.

[18] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[19] A. Awad, A. Basu, S. Blagodurov, Y. Solihin, and G. H. Loh. Avoiding TLB shootdowns through self-invalidating TLB entries. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 273–287, 2017.

[20] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis. *ACM SIGARCH Computer Architecture News*, 38(3):26–36, 2010.

[21] I. Baldini, S. J. Fink, and E. Altman. Predicting GPU performance from CPU runs using machine learning. In *Proceedings of International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 254–261, 2014.

[22] R. A. Bheda, J. A. Poovey, J. G. Beu, and T. M. Conte. Energy efficient phase change memory based main memory for future high performance systems. In *Proceedings of International Green Computing Conference and Workshops (IGCC)*, pages 1–8, 2011.

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

[24] A. Biswas, P. Racunas, J. Emer, and S. Mukherjee. Computing accurate AVFs using ACE analysis on performance models: A rebuttal. *IEEE Computer Architecture Letters (CAL)*, 7(1):21–24, 2008.

[25] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh1, D. McCauley, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb. Die stacking (3D) microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–479, 2006.

[26] S. M. Blackburn and K. 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)*, pages 22–32, 2008.

[27] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 137–146, 2004.

[28] S. M. Blackburn, P. Cheng, and K. 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)*, pages 25–36, 2004.

[29] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, 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. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-oriented*

Programming Systems, Languages, and Applications (OOPSLA), pages 169–190, 2006.

[30] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, 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. Stefanovik, 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.

[31] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *ACM SIGARCH computer architecture news*, 25(3):13–25, 1997.

[32] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52(4.5):449–464, 2008.

[33] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 225–236, 2012.

[34] T. E. Carlson, W. Heirman, and L. 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)*, pages 1–12, 2011.

[35] T. E. Carlson, W. Heirman, and L. Eeckhout. Sampled simulation of multi-threaded applications. In *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12, 2013.

[36] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3):1–25, 2014.

[37] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout. Barrierpoint: Sampled simulation of multi-threaded applications. In *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12, 2014.

[38] N. Chatterjee, M. Shevgoor, R. Balasubramonian, A. Davis, Z. Fang, R. Illikkal, and R. Iyer. Leveraging heterogeneity in dram main memories to accelerate critical word access. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 13–24, 2012.

[39] Y. Cheng, A. Ma, and M. Zhang. Accurate and simplified prediction of l2 cache vulnerability for cost-efficient soft error protection. *IEICE transactions on Information and Systems*, 95(1):56–66, 2012.

[40] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat. FPGA-accelerated simulation technologies (FAST): Fast, full-system, cycle-accurate simulators. In *Proceedings of the 40th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 249–261, 2007.

[41] J. Choi, T. Shull, M. J. Garzaran, and J. Torrellas. Shortcut: Architectural support for fast object access in scripting languages. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 494–506, 2017.

[42] C. Chou, A. Jaleel, and M. K. Qureshi. CAMEO: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2014.

[43] C. Chou, A. Jaleel, and M. K. Qureshi. BEAR: Techniques for mitigating bandwidth bloat in gigascale DRAM caches. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, page 198–210, 2015.

[44] C. Chou, A. Jaleel, and M. Qureshi. BATMAN: Techniques for maximizing system bandwidth of memory systems with stacked-DRAM. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, pages 268–280, 2017.

[45] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-Chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 225–236, 2010.

[46] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, page 128–137, 1994.

[47] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of International Conference on Computer Design. VLSI in Computers and Processors*, pages 468–477, 1996.

[48] S. De Pestel, S. Van den Steen, S. Akram, and L. Eeckhout. RPPM: Rapid performance prediction of multithreaded workloads on multicore processors. In *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 257–267, 2019.

[49] T. J. Dell. A white paper on the benefits of chipkill-correct ECC for PC server main memory. *IBM Microelectronics division*, 11(1-23):5–7, 1997.

[50] Y. Demchenko, C. de Laat, and P. Membrey. Defining architecture components of the big data ecosystem. In *Proceedings of International Conference on Collaboration Technologies and Systems (CTS)*, pages 104–112, 2014.

[51] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, SC, pages 1–11, 2010.

[52] K. Du Bois, J. B. Sartor, S. Eyerman, and L. 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)*, pages 355–372, 2013.

[53] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. De Bosschere. Statistical simulation: adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, 2003.

[54] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, et al. Asim: A performance model framework. *Computer*, 35(2):68–76, 2002.

[55] P. G. Emma. Understanding some simple processor-performance limits. *IBM journal of Research and Development*, 41(3):215–232, 1997.

[56] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE micro*, 28(3):42–53, 2008.

[57] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. 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)*, pages 175–184, 2006.

[58] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems (TOCS)*, 27(2):1–37, 2009.

[59] S. Eyerman, W. Heirman, Y. Demir, K. Du Bois, and I. Hur. Projecting performance for PIUMA using down-scaled simulation. In *Proceedings of IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2020.

[60] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy. Unified instruction/translation/data (UNITD) coherence: One protocol to rule them all. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[61] D. Fisher, R. DeLine, M. Czerwinski, and S. Drucker. Interactions with big data analytics. *Interactions*, 19(3):50–59, 2012.

[62] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev. Demystifying magic: High-level low-level programming. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, pages 81–90, 2009.

[63] W. J. Gallagher and S. S. P. Parkin. Development of the magnetic tunnel junction MRAM at IBM: From first junctions to a 16-Mb MRAM demonstrator chip. *IBM Journal of Research and Development*, 50(1): 5–23, 2006.

[64] T. Gao, K. Strauss, S. M. Blackburn, K. S. McKinley, D. Burger, and J. Larus. Using managed runtime systems to tolerate holes in wearable memories. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 297–308, 2013.

[65] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[66] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. *ACM SIGMETRICS Performance Evaluation Review*, 21(1):146–157, 1993.

[67] Google Cloud Platform. Cloud functions - serverless environment to build and connect cloud services. `https://cloud.google.com/functions/`.

[68] T. Grass, C. Allande, A. Armejach, A. Rico, E. Ayguadé, J. Labarta, M. Valero, M. Casas, and M. Moreto. MUSA: a multi-level simulation approach for next-generation HPC machines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 526–537, 2016.

[69] M. Gupta, V. Sridharan, D. Roberts, A. Prodromou, A. Venkat, D. Tullsen, and R. Gupta. Reliability-aware data placement for heterogeneous memory architecture. In *Proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 583–595, 2018.

[70] G. H. Loh and M. D. Hill. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 454–564, 2011.

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

[72] A. Haghdoost, H. Asadi, and A. Baniasadi. System-level vulnerability estimation for data caches. In *Proceedings of the 16th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 157–164, 2010.

[73] R. W. Hamming. Error detecting and error correcting codes. *The Bell system technical journal*, 29(2):147–160, 1950.

[74] A. Hartstein and T. R. Puzak. The optimum pipeline depth for a microprocessor. *ACM Sigarch Computer Architecture News*, 30(2):7–13, 2002.

[75] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 114–122, 2006.

[76] M.-Y. Hsiao. A class of optimal minimum odd-weight-column SEC-DED codes. *IBM Journal of Research and Development*, 14(4):395–401, 1970.

[77] Y. Huai, R. Lee, S. Zhang, C. H. Xia, and X. Zhang. DOT: A matrix model for analyzing, optimizing and deploying software for big data analytics in distributed systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*, pages 1–14, 2011.

[78] J. Huang and M. 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)*, pages 53–72, 2013.

[79] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. 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)*, pages 69–80, 2004.

[80] IBM. Cloud functions. `https://www.ibm.com/cloud/functions`.

[81] IBM. Enhancing IBM netfinity server reliability: IBM chipkill memory. Technical report, 2000. URL `http://www.ece.umd.edu/courses/enee759h.S2003/references/chipkill_white_paper.pdf`.

[82] E. Ipek, S. McKee, R. Caruana, d. B. R., and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 195–206, 2006.

[83] ITRS. Internatial technology roadmap for semiconductors: ASSEMBLY AND PACKAGING, 2005.

[84] P. J. Nair, D. A. Roberts, and M. K. Qureshi. Citadel: Efficiently protecting stacked memory from large granularity failures. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 51–62, 2014.

[85] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. CMP$im: A pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, pages 28–36, 2008.

[86] JEDEC. High bandwidth memory. `https://www.jedec.org/standards-documents/docs/jesd235a`.

[87] H. Jeon, G. H. Loh, and M. Annavaram. Efficient RAS support for die-stacked DRAM. In *Proceedings of the International Test Conference (ITC)*, pages 1–10, 2014.

[88] D. Jevdjic, S. Volos, and B. Falsafi. Die-stacked DRAM caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 404–415, 2013.

[89] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi. Unison Cache: A scalable and effective die-stacked DRAM cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 25–37, 2014.

[90] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally. A detailed and flexible cycle-accurate network-on-chip simulator. In *Proceedings of IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 86–96, 2013.

[91] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. CHOP: Adaptive filter-based DRAM caching for CMP server platforms. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[92] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. Chop: Integrating dram caches for cmp server platforms. *IEEE micro*, 31(1):99–108, 2010.

[93] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 223–234, 2012.

[94] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., 1996.

[95] R. Jongerius, A. Anghel, G. Dittmann, G. Mariani, E. Vermij, and H. Corporaal. Analytic multi-core processor model for fast design-space exploration. *IEEE Transactions on Computers (TC)*, 67(6):755–770, 2017.

[96] M. K. Qureshi and G. H. Loh. Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-Tags with a simple and practical design. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 235–246, 2012.

[97] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 338–349, 2004.

[98] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, pages 361–372, 2014.

[99] K. Krewell. Intel's McKinley comes into view. *Microprocessor Report*, 15 (10):1, 2001.

[100] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267, 2013.

[101] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 81–92, 2003.

[102] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 64–75, 2004.

[103] E. Larson, S. Chatterjee, and T. M. Austin. MASE: A novel infrastructure for detailed microarchitectural modeling. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 1–9, 2001.

[104] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 185–194, 2006.

[105] B. C. Lee and D. M. Brooks. Illustrative design space studies with microarchitectural regression models. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 340–351, 2007.

[106] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 249–258, 2007.

[107] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Leet. A fully associative, tagless DRAM cache. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 211–222, 2015.

[108] D. J. Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge university press, 2005.

[109] W. Liu, S. Akram, J. B. Sartor, and L. Eeckhout. Reliability-Aware Garbage Collection for Hybrid HBM-DRAM Memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(1):1–25, 2021.

[110] W. Liu, W. Heirman, S. Eyerman, S. Akram, and L. Eeckhout. Scale-Model Simulation. *IEEE Computer Architecture Letters (CAL)*, 20(2): 175–178, 2021.

[111] W. Liu, W. Heirman, S. Eyerman, S. Akram, and L. Eeckhout. Scale-Model Architectural Simulation. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022, Accepted.

[112] X. Liu, D. Roberts, R. Ausavarungnirun, O. Mutlu, and J. Zhao. Binary star: Coordinated reliability in heterogeneous memory systems for high performance and scalability. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 807–820, 2019.

[113] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[114] E. McClusky and S. Mitra. Fault tolerance. *Computer Science Handbook 2ed. ed. AB Tucker. CRC Press Amazon EC2 home page, http://aws.amazon.com/ec2*, 2004.

[115] M. Meyer. A true hardware read barrier. In *Proceedings of the 5th International Symposium on Memory Management (ISMM)*, pages 3–16, 2006.

[116] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan. Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management. *IEEE Computer Architecture Letters (CAL)*, 11(2):61–64, 2012.

[117] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 415–526, 2015.

[118] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 2–10, 1999.

[119] Micron. Tn-41-01: Calculating memory system power for DDR3. `https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tn41_01ddr3_power.pdf`, 2007.

[120] Microsoft Azure. Azure functions serverless architecture. `https://azure.microsoft.com/en-us/services/functions`.

[121] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[122] J. C. Mogul, E. Argollo, M. A. Shah, and P. Faraboschi. Operating system support for NVM+DRAM hybrid main memory. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems (HotOS)*, pages 4–14, 2009.

[123] A. Mohammad, U. Darbaz, G. Dozsa, S. Diestelhorst, D. Kim, and N. S. Kim. dist-gem5: Distributed simulation of computer clusters. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 153–162, 2017.

[124] P. Montesinos, W. Liu, and J. Torrellas. Using register lifetime predictions to protect register files against soft errors. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 286–296, 2007.

[125] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of the 16th USENIX Security Symposium on USENIX Security Symposium (SS)*, pages 1–18, 2007.

[126] S. Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers, 2008.

[127] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 29–40, 2003.

[128] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th International Symposium on Microarchitecture (MICRO)*, pages 146–160, 2007.

[129] P. Nagpurkar, H. W. Cain, M. Serrano, J.-D. Choi, and C. Krintz. Call-chain software instruction prefetching in J2EE server applications. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 140–149, 2007.

[130] P. J. Nair, D. A. Roberts, and M. K. Qureshi. FaultSim: A fast, configurable memory-reliability simulator for conventional and 3D-stacked systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):1–24, 2015.

[131] A. Naithani, S. Eyerman, and L. Eeckhout. Reliability-aware scheduling on heterogeneous multicore processors. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 397–408, 2017.

[132] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 349–365, 2016.

[133] NVIDIA Corp. NVIDIA Pascal Architecture. `https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/`.

[134] M. Omana, G. Papasso, D. Rossi, and C. Metra. A model for transient fault propagation in combinatorial logic. In *Proceedings of the 9th IEEE On-Line Testing Symposium (IOLTS)*, pages 111–115, 2003.

[135] M. Oskin and G. H. Loh. A software-managed approach to die-stacked DRAM. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 188–200, 2015.

[136] M. O'Connor. Highlights of the high-bandwidth memory (HBM) standard. `https://www.cs.utah.edu/thememoryforum/mike.pdf`, 2014.

[137] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–11, 2010.

[138] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2013.

[139] M. Pavlovic, N. Puzovic, and A. Ramirez. Data placement in HPC architectures with heterogeneous off-chip memory. In *Proceedings of the 31st IEEE International Conference on Computer Design (ICCD)*, pages 193–200, 2013.

[140] J. Pawlowski. Hybrid memory cube (HMC): Breakthrough DRAM performance with a fundamentally re-architected DRAM subsystem. In *Hot Chips*, volume 23, 2011.

[141] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis. Exploring the performance benefit of hybrid memory system on HPC environments. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 683–692, 2017.

[142] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 244–255, 2003.

[143] S. Phadke and S. Narayanasamy. MLP aware heterogeneous memory system. In *Proceedings of Design, Automation & Test in Europe (DATE)*, pages 1–6, 2011.

[144] B. Piccart, A. Georges, H. Blockeel, and L. Eeckhout. Ranking commercial machines through data transposition. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, pages 3–14, 2011.

[145] A. Prodromou, M. Meswani, N. Jayasena, G. Loh, and D. M. Tullsen. MemPod: A clustered architecture for efficient and scalable migration in flat address space multi-level memories. In *Proceedings of the 23rd IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 433–444, 2017.

[146] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 126–136, 2015.

[147] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 85–95, 2011.

[148] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel. Reliable software for unreliable hardware: Embedded code generation aiming at reliability. In *Proceedings of the 9th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 237–246, 2011.

[149] S. Rehman, A. Toma, F. Kriebel, M. Shafique, J.-J. Chen, and J. Henkel. Reliable code generation and execution on unreliable hardware under joint functional and timing reliability considerations. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 273–282, 2013.

[150] A. Rico, F. Cabarcas, A. Quesada, M. Pavlovic, A. J. Vega, C. Villavieja, Y. Etsion, and A. Ramirez. Scalable simulation of decoupled accelerator architectures. *Universitat Politecnica de Catalunya, Tech. Rep. UPCDAC-RR-2010-14*, 2010.

[151] A. Rodchenko, C. Kotselidis, A. Nisbet, A. Pop, and M. Luján. MaxSim: A simulation platform for managed applications. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 141–152, 2017.

[152] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: Challenges in and avenues for CMP scaling. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, pages 371–382, 2009.

[153] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 29–40, 2003.

[154] A. Sabu, H. Patil, W. Heirman, and T. E. Carlson. Looppoint: Checkpoint-driven sampled simulation for multi-threaded applications. In *Proceedings of the 28th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–15, 2022.

[155] N. Sachindran and J. E. B. Moss. Mark-Copy: Fast copying GC with less space overhead. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 326–343, 2003.

[156] D. Sanchez and C. Kozyrakis. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the Inter-*

national Symposium on Computer Architecture (ISCA), pages 475–486, 2013.

[157] K. Sangaiah, M. Lui, R. Jagtap, S. Diestelhorst, S. Nilakantan, A. More, B. Taskin, and M. Hempstead. Synchrotrace: Synchronization-aware architecture-agnostic traces for lightweight multicore simulation of CMP and HPC workloads. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(1):1–26, 2018.

[158] J. B. Sartor, W. Heirman, S. M. Blackburn, L. Eeckhout, and K. S. McKinley. Cooperative cache scrubbing. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, pages 15–26, 2014.

[159] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: A large-scale field study. *ACM SIGMETRICS Performance Evaluation Review*, 37(1):193–204, 2009.

[160] R. Shahriyar, S. M. Blackburn, X. Yang, and K. 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)*, pages 93–110, 2013.

[161] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 45–57, 2002.

[162] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. *ACM SIGARCH Computer Architecture News*, 31(2):336–349, 2003.

[163] T. Shull, J. Huang, and J. Torrellas. AutoPersist: An easy-to-use Java NVM framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 316–332, 2019.

[164] J. Sim, G. H. Loh, H. Kim, M. OConnor, and M. Thottethodi. A mostly-clean DRAM cache for effective hit speculation and self-balancing dispatch. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 247–257, 2012.

[165] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim. Transparent hardware management of stacked DRAM as part of memory. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 13–24, 2014.

[166] K. Singh, M. Bhadauria, and S. A. McKee. Real time power estimation and thread scheduling via performance counters. *ACM SIGARCH Computer Architecture News*, 37(2):46–55, 2009.

[167] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch prediction, instruction-window size, and cache size: Performance trade-offs and simulation techniques. *IEEE Transactions on Computers (TC)*, 48 (11):1260–1281, 1999.

[168] S. Smith. Announcing Oracle Functions. `https://blogs.oracle.com/cloud-infrastructure/announcing-oracle-functions`, 2018.

[169] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46, 2016.

[170] V. Sridharan and D. Liberty. A study of DRAM failures in the field. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2012.

[171] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 297–310, 2015.

[172] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 639–650, 2013.

[173] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 253–264, 2009.

[174] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi. Rethinking DRAM design and organization for energy-constrained multi-cores. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, pages 175–186, 2010.

[175] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys (CSUR)*, 29(2):128–170, 1997.

[176] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the 1st ACM SIGSOFT-/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE)*, pages 157–167, 1984.

[177] D. Ungar and F. Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(1):1–27, 1992.

[178] K. Van Craeynest and L. Eeckhout. The multi-program performance model: debunking current practice in multi-core simulation. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, pages 26–37, 2011.

[179] A. Venkat and D. M. Tullsen. Harnessing ISA diversity: Design of a Heterogeneous-ISA chip multiprocessor. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, page 121–132, 2014.

[180] A. Venkat, S. Shamasunder, H. Shacham, and D. M. Tullsen. HIPStR: Heterogeneous-ISA program state relocation. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 727–741, 2016.

[181] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal. DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 340–349, 2011.

[182] C. Wang, H. Cui, T. Cao, J. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, and G. H. Xu. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 347–362, 2019.

[183] S. Wang. Characterizing system-level vulnerability for instruction caches against soft errors. In *Proceedings of IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 356–363, 2011.

[184] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 264–275, 2004.

[185] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-l. Lu. Reducing cache power with low-cost, multi-bit error-correcting codes. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, pages 83–93, 2010.

[186] C. Wimmer. Initialize once, start fast: Application initialization at build time. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–29, 2019.

[187] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding. Data mining with big data. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 26(1): 97–107, 2014.

[188] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 84–97, 2003.

[189] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley. Why nothing matters: The impact of zeroing. In *Proceedings of the ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 307–324, 2011.

[190] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking. Barriers reconsidered, friendlier still! In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 37–48, 2012.

[191] V. Young, C. Chou, A. Jaleel, and M. Qureshi. ACCORD: Enabling associativity for gigascale DRAM caches by coordinating way-install and way-prediction. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 328–339, 2018.

[192] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 23–34, 2007.

[193] J. Zhao, G. Sun, G. H. Loh, and Y. Xie. Energy-efficient GPU design with reconfigurable in-package graphics memory. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, pages 403–408, 2012.

[194] Y. Zhao, J. Shi, K. Zheng, H. Wang, H. Lin, and L. Shao. Allocation wall: A limiting factor of Java applications on emerging multi-core platforms. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 361–376, 2009.

Hybrid memories in modern multi-core processors require stronger protection against soft errors.