UNIVERSITY OF CALIFORNIA SAN DIEGO

Achieving Accurate Predictions of Future Events Under Hardware Heterogeneity

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Andreas Prodromou

2019

The Dissertation of Andreas Prodromou is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego

2019

DEDICATION

To my family and friends, whom I’ve missed terribly during this endeavor.

EPIGRAPH

Prediction is very difficult, especially if it’s about the future.

Nils Bohr, Physics Nobel Prize winner

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would first like to thank my advisor, professor Dean Tullsen for giving me this incredible opportunity. None of this would be possible without him and his support over the years. I would also like to thank Dr. Ashish Venkat who has been a great collaborator. Our brainstorming sessions with both Dean and Ashish have contributed greatly towards achieving this goal. I want to take this opportunity to thank all my research collaborators and co-authors over the years, as well as my PhD committee members.

I owe a debt of gratitude to my family. My parents, my brother and sisters, and my grandmother have always been there for me. Knowing I could always turn to them with any problem makes everything possible. My mother, Anna Prodromou was my first teacher. Both figuratively, but literally as well during 4^{th} grade. Always loving, patient and supportive. I couldn't have asked for a better mom. My dad, Demetres Prodromou has also been an amazing parent. One of the smartest and kindest people I know, always setting the example for me and my siblings. His dedication to his family and his principles enlightens the lives of everybody around him. I specifically remember three landmark moments in my academic career, where I was ready to give up and my dad's advice and encouragement changed my mind. I am forever grateful to both of them.

I would also like to thank my brother Nicolas, and my sisters Zoe and Thekla. I know I can always count on them and they can count on me. As they are now starting their own families and careers, I wish them all the best and I am very happy for them.

My grandmother, Zoe Vourgia is part of some of my fondest childhood memories. She's been a pillar of support and wisdom during my entire life. I have always admired her patience and perseverance and she's always encouraged me to strive for perfection. I cannot thank her enough for her support during my graduate studies.

I would like to thank all my childhood friends from Cyprus: Nicolas Petrou, Simos Pafitis, Koullis Hadjipavlou, Andreas Charalambous, Vasilis Vasiliou, Andreas, Eva, and Maria Dikomiti. Even with all the distance between us, these friendships never deteriorated. Every

time I meet with them feels as if I was never gone. Every time I talk to any one of them while away from home fills me with incredible joy.

The graduate student life is full of challenges, which could be insurmountable without the friendships developed over the years in San Diego. While too many to mention by name, all of us have been through a lot together. It's not a coincidence that on days that are traditionally spent with family such as Thanksgiving and Christmas, we spent them together as a big family away from home.

I would also like to extend my gratitude to my closest friends in San Diego: Dimosthenis Giamouridis and Constantinos Zarifis: The best company to relax with at the end of all the long days of graduate school. The prevailing silliness and lightness of our interactions was the best medicine to combat the seemingly endless stress of research and life as a graduate student. I consider both of them brothers.

Special thanks goes to Bree Gomez for her immense support during the most challenging period of my life. Her cheerfulness and energy has always had a way to make me forget all my troubles. Thank you and I wish you all the best Bree.

Last but not least, I'd like to thank Jennifer Verdier. I can confidently say I wouldn't be who I am today if it wasn't for Jen. What started with a series of incredible coincidences about 7000 miles away and over a decade ago, transformed into the adventure of a lifetime. As with all great adventures, the ups were accompanied by occasional downs and the future remains to be seen. I consider myself lucky to have met her and honored to call her my friend. Thanks Jen!

Chapter 2, in full, is a reprint of the material as it appears in the proceedings of HPCA 2017. Prodromou, Andreas; Meswani, Mitesh; Jayasena, Nuwan; Loh, Gabriel; Tullsen, Dean M., MemPod: A Clustered Architecture for Efficient and Scalable Migration in Flat Address Space Multi-level Memories, 23^{rd} IEEE Symposium on High Performance Computer Architecture (HPCA), February, 2017. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in part, is a reprint of the material as it appears in the proceedings of PMAM

2019. Prodromou, Andreas; Venkat, Ashish; Tullsen, Dean M., Deciphering Predictive Schedulers for Heterogeneous-ISA Multicore Architectures, 10^{th} International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM), held in conjunction with PPoPP 2019. Chapter 3 further contains a partial reprint of the material as it appears in the proceedings of SAMOS 2019. Prodromou, Andreas; Venkat, Ashish; Tullsen, Dean M., Platform-Agnostic Learning-Based Scheduling, 19^{th} International Conference on Embedded Computer Systems: Architecture MOdeling and Simulation (SAMOS). The dissertation author was the primary investigator and author of both these papers.

Chapter 4, in full, is currently being prepared for submission for publication of the material. Prodromou, Andreas; Venkat, Ashish; Tullsen, Dean M., Agon: A Scalable Competitive Scheduler. The dissertation author was the primary investigator and author of this material.

VITA

2005-2007	Military service, Greek Cypriot National Guard
2009	Undergraduate Research Intern, KIOS Research Center, Cyprus
2010	Research Intern, Multicore Computer Architecture Laboratory (multiCAL), Cyprus
2011	Bachelor of Science, University of Cyprus, Cyprus
2011-2013	Research Assistant, ECE Department, University of Cyprus, Cyprus
2013	Master of Science, University of Cyprus, Cyprus
2015	Co-op Intern, Advance Micro Devices (AMD), Austin, Texas
2016	Co-op Intern, Advance Micro Devices (AMD), Austin, Texas
2018	Deep Learning Architecture Intern, NVIDIA, Santa Clara, California
2013-2019	Research Assistant, University of California San Diego
2013-2019	Teaching Assistant, University of California San Diego
2019	Doctor of Philosophy, University of California San Diego

PUBLICATIONS

Prodromou Andreas, Ashish Venkat, Dean M. Tullsen. "Agon: A Scalable Competitive Scheduler." *Under Review*, 2019

Prodromou Andreas, Ashish Venkat, Dean M. Tullsen. "Platform-Agnostic Learning-Based Scheduling." *SAMOS*, 2019

Prodromou Andreas, Ashish Venkat, Dean M. Tullsen. "Deciphering Predictive Schedulers for Heterogeneous-ISA Multicore Architectures." *PMAM@PPoPP*, 2019

Manish Gupta, Vilas Sridharan, David Roberts, **Prodromou Andreas**, Ashish Venkat, Dean M. Tullsen, Rajesh K. Gupta. "Reliability-Aware Data Placement for Heterogeneous Memory Architecture." *HPCA*, 2018

Jee Ho Ryoo, Mitesh R. Meswani, **Prodromou Andreas**, Lizy K. John. "SILC-FM: Subblocked InterLeaved Cache-Like Flat Memory Organization." *HPCA*, 2017

Prodromou Andreas, Mitesh R. Meswani, Nuwan Jayasena, Gabriel H. Loh, Dean M. Tullsen. "MemPod: A Clustered Architecture for Efficient and Scalable Migration in Flat Address Space Multi-level Memories." *HPCA*, 2017

Kypros Chrysanthou, Panayiotis Englezakis, **Prodromou Andreas**, Andreas Panteli, Chrysostomos Nicopoulos, Yiannakis Sazeides, Giorgos Dimitrakopoulos. "An Online and Real-Time

Fault Detection and Localization Mechanism for Network-on-Chip Architectures." *TACO 13*, 2016

Dragomir Milojevic, Sachin Idgunji, Djordje Jevdjic, Emre zer, Pejman Lotfi-Kamran, Andreas Panteli, **Prodromou Andreas**, Chrysostomos Nicopoulos, Damien Hardy, Babak Falsafi, Yiannakis Sazeides. "Thermal characterization of cloud workloads on a power-efficient server-on-chip." *ICCD* 2012

Prodromou Andreas, Andreas Panteli, Chrysostomos Nicopoulos, and Yiannakis Sazeides. "Nocalert: An on-line and real-time fault detection mechanism for network-on-chip architectures." *MICRO*, 2012

ABSTRACT OF THE DISSERTATION

Achieving Accurate Predictions of Future Events Under Hardware Heterogeneity

by

Andreas Prodromou

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2019

Professor Dean M. Tullsen, Chair

Heterogeneous hardware is becoming increasingly available in modern hardware, while research breakthroughs enforce the expectation that heterogeneity will keep increasing in the future. Significant gains can be achieved via appropriate utilization of heterogeneity, in terms of performance and power consumption, however, poor utilization can have a detrimental effect. Intelligent scheduling and resource management is a crucial challenge we need to overcome in order to harvest the full potential of heterogeneous hardware. As systems become larger and include greater levels of hardware diversity, the importance of intelligent scheduling and resource management is further accentuated.

This dissertation presents techniques that aid the process of scheduling and resource management in the presence of heterogeneous hardware, via accurately predicting upcoming runtime events. With a proactive and accurate view of the near future, schedulers can utilize the underlying hardware more efficiently, and fully take advantage of the available benefits.

By adapting a majority element heuristic, this dissertation significantly improves the accuracy of predicting memory addresses about to be accessed, while reducing prediction-related costs by a factor of ten thousand compared to previously proposed predictive approaches. Coupled with novel microarchitectural modifications, accurate address predictions are shown to improve the performance of heterogeneous memory architectures.

Machine learning-based performance predictors are further presented, capable of predicting a program's performance when executed on a given general-purpose core. Trained to model the subtleties of the interaction between hardware and software, these predictors are capable of generating highly accurate predictions even for cores with varied Instruction Set Architectures. Utilizing these performance predictions for job scheduling, is shown to improve overall system performance. The trained predictors are further examined and interpreted in order to visualize the correlations between features picked up and amplified during training.

Finally, this dissertation demonstrates that scheduling algorithms cannot guarantee deriving an optimal schedule during realistic execution scenarios due to the underlying hardware heterogeneity, the wide range of runtime requirements of software, as well as prediction error from performance predictors. In response, deep neural networks are trained to select one scheduling approach from a list of options with varied overheads and correctness guarantees. The scheduling approach chosen, is the one which will most likely return the highest-performance schedule with the lowest overhead, given a particular instance of the job-to-core assignment problem.

Chapter 1

Introduction

Recent years have seen an increase in interest in heterogeneous hardware, both in industry and in the research literature. The requirements of modern software have grown more diverse and consequently, homogeneous general-purpose processors designed to perform well on the average case, can prove over- or under-provisioned for executing one given program. In response to the ever-growing computational needs of the modern world, hardware heterogeneity allows for varied hardware to coexist in a same processor, providing a wide range of computing options. Heterogeneous memory architectures for example, can accelerate software by serving data from the memory with the highest bandwidth. Compute-bound programs which can utilize the majority of available resources can be assigned on big high-performance cores. Memory-bound software that spends most of its execution time idle waiting for data retrieval can run on slower, power efficient cores.

Hardware heterogeneity creates the potential for significant benefits, such as increased performance and reduced power consumption. Harvesting its full potential however, is challenging, as it requires effectively assigning diverse tasks with constantly changing resource needs, to the best combination of hardware engines at any point in time. The larger the system and the greater the level of diversity, the more difficult it is to find the best solution. Homogeneous systems do not present this challenge, as all execution options are expected to result in roughly equivalent performance and power consumption. In contrast, heterogeneous systems need careful scheduling, as the effects of poor assignment decisions can waste resources, or negatively impact performance.

This dissertation focuses on designing high-quality schedulers and resource managers for heterogeneous architectures and systems, by predicting near-future runtime events. Throughout this document, a variety of predictive approaches are proposed, presented, and evaluated, ranging from simple heuristics, to trained machine learning models, and deep neural networks. These predictors are shown to improve three aspects of execution on heterogeneous hardware: First, the performance of main memory improves by accurately predicting which memory addresses a running program is about to access. This information allows a memory manager to transparently migrate data between the available memory technologies that coexist in a system, maximizing the amount of data served from the faster memory. Second, predictors are shown to be able to learn, through training, the interaction between software and hardware and accurately predict the runtime performance of a program-core pair. This information is later used by a scheduler to generate a program-to-core assignment that maximizes overall system performance. Finally, deep neural network models are trained to select the best scheduling algorithm to generate this program-to-core assignment from a selection of schedulers that vary in their overhead and system performance derived from the assignments they generate.

Heterogeneity in a general-purpose processor can take a number of forms:

Heterogeneous Memory Architectures [84, 90, 69, 72, 27]: These systems include two (or more) different main memory technologies and organization schemes, such as non-volatile memories (NVDRAM) [73, 91] like Phase-Changing Memory (PCM) [86], High-Bandwidth Memory (HBM) [49, 50] and Hybrid Memory Cube (HMC) [85]. In systems with Heterogeneous Memory Architectures, the address space can be managed as a contiguous range of addresses that spans the capacity of all existing memories combined (flat address space) [72, 96, 27], or with one memory used as a large, Last-Level Cache (LLC) [90, 28, 38, 77, 65, 48].

Heterogeneous Multicore Architectures [62, 63]: These architectures allow for the co-packaging of heterogeneous general-purpose processing cores that differ in their micro-architectures. For example, one core can be high-performance, have a large instruction window and execute instructions out-of-order, while another core can be low-performance, with a simple in-order, power-efficient pipeline. Processor manufacturers are already offering such heterogeneous multicore products [37, 3, 4, 42, 1, 2]. Heterogeneous multicore architectures can include diverse cores, however, these cores must implement the same Instruction Set Architecture (ISA). The ISA is the set of low-level instructions a processor understands (i.e. can execute). Two cores with the same ISA can execute the same program without recompiling it.

Heterogeneous-ISA Multicore Architectures [32, 108, 106, 10, 105]: Heterogeneous-ISA multicore architectures allow for the divergence of not only the micro-architecture of cores, but also their ISAs. Due to the incompatibility between the cores' ISAs, freely migrating an executing program between cores becomes especially challenging. Prior work has developed a compilation and runtime infrastructure to allow for such migrations to take place [32, 108]. Furthermore, prior work has demonstrated that the coexisting ISAs can be derived from modifications on a single ISA [106], partly alleviating some of the runtime complexity, but also alleviating concerns about the market viability of such products, since otherwise it could only be done if major competing manufacturers agree to collaborate and share their ISAs.

Accelerator-based heterogeneity: This type of heterogeneity utilizes highly optimized hardware, that can only execute narrow sets of applications: Application Specific Integrated Circuits (ASICs) [97] are designed to execute a single task with the utmost efficiency. Field Programmable Gate Arrays (FPGAs) [18] also accelerate specific tasks, and further provide the ability to reprogram as needed for a different task. Graphics Processing Units (GPUs) can extract massive performance gains from highly-parallelizable software [80]. As such they are used in the majority of scientific High Performance Computing (HPC) applications [99, 40, 56, 95] and populate the world's fastest supercomputers alongside general-purpose processors [43]. Typically, a general-purpose core is required to drive these accelerators, by transmitting necessary data, and coordinating execution. Due to the lack of generality of these accelerators, this dissertation focuses on systems with all-general-purpose hardware (Heterogeneous Memory and heterogeneous(-ISA) multicore architectures)

1.1 Achieving Accurate Predictions

Successful and accurate predictions require an understanding of the underlying environment and the context under which the predictions occur. For example, predictive approaches for heterogeneous memories do not necessarily adapt well to the context of heterogeneous multicores. This section introduces the key concepts that contribute to high-quality predictions for each form of heterogeneity.

1.1.1 Heterogeneous Memory Architectures

In the presence of different main memory technologies, the problem of resource allocation reduces to that of data partitioning. In other words, a successful memory controller will ensure that data is physically stored in the appropriate memory type, in such a way as to accelerate program execution, reduce power consumption, or improve the reliability of data.

A simple example assumes the presence of two memories: a small and fast, and a slower and larger. Ideally, all data touched by a program would be serviced from the fast memory, but its limited capacity imposes a challenge. To overcome, a memory controller can attempt to predict the data about to be accessed and transparently transfer it into the fast memory during times of low memory activity.

The problem is similar to that of data prefetching for cache memories. Main memory however, operates at larger granularity than caches, rendering many pre-fetching approaches unsuccessful when deployed on main memories. A cache line is typically 64B, while a main memory page is 2KB and contains multiple cache lines.

Chapter 2 of this dissertation presents “MemPod”; a proposal towards more efficient Heterogeneous Memory Architectures. MemPod features a novel clustered architecture which groups sets of memory channels of the two participating kinds of memories into independent “Pods”. This divide-and-conquer approach simplifies bookkeeping overheads and allows for better scalability to larger memories with potentially more channels. Besides the novel microarchitectural proposal, MemPod also features a powerful and lightweight predictive unit which is based on the “Majority Element Algorithm” (MEA) [55, 22].

The exploration presented in Chapter 2 reveals that in order to design an accurate predictor, we must strike the right balance between spatial (accessing neighboring data; for example iterating through an array of values), and temporal (re-accessing the same data soon after it was accessed) locality. Programs exhibit a wide spectrum of runtime behavior in regards to data access. Some are highly predictable and follow strict patterns, while others are more chaotic and consequently less predictable. The MEA predictor successfully balances the two kinds of locality: While it assigns significant weight to those memory pages that have been accessed heavily, it strikes a balance by also assigning high weight to the few most recently accessed pages. Compared to traditionally used prediction techniques, this predictor is shown to be more flexible and as a result more accurate in a wider range of applications. Furthermore, by adapting a lightweight bookkeeping scheme originally proposed for big data management, the cost of main memory predictions reduces by more than ten thousand times compared to other techniques.

This predictor and its accompanying proposed microarchitectural modifications, has been published in the 23rd Symposium on High Performance Computer Architecture (HPCA 2017). Furthermore, in collaboration with Advanced Micro Devices (AMD), part of this work has also been patented on May, 2019 (Patent No: US 10,282,292 B2).

1.1.2 Heterogeneous(-ISA) Multicore Architectures

Heterogeneous and heterogeneous-ISA multicore architectures present a different problem than memories: Given a number of cores (let’s assume N cores) and an equal number of workloads to be executed, we want to find a one-to-one mapping of jobs to cores, such that it maximizes the system’s performance.

This dissertation proposes, evaluates and presents performance predictors, capable of predicting the runtime performance of arbitrary workload-core pairs. These predictors generate a square, $N \times N$ matrix of predictions, which is then passed to the scheduler for the last step of creating a job assignment.

The problem of performance prediction for heterogeneous-ISA architectures presents a new set of challenges compared to memory address prediction: Software and hardware interacts in ways that are not easily predictable. For example, if a change in hardware (say doubling the L1 cache size) improves the performance of program A by 50%, we cannot expect the same impact on program B. Depending on B's runtime behavior, this particular change in hardware can present no impact whatsoever, or affect it at a different degree. Software-hardware interaction cannot be well fitted with a simple function, instead more capable predictive structures need to be deployed.

Finally, allowing the ISA to differ between cores, raises the difficulty of this prediction problem, since software no longer behaves the same way when (compiled for and) executed on cores of different ISAs.

In Chapter 3, we investigate the capabilities of performance predictors in a heterogeneous-ISA setting, as well as the predictors' effects on scheduler quality. We follow an unbiased feature selection methodology to identify the optimal set of features for this task, instead of pre-selecting features before training. We propose metrics that bridge the gap between traditional prediction accuracy metrics and a scheduler's performance. We further present our evaluation methodology, which was meticulously designed with this study in mind, and finally, we incorporate our findings in ML-based schedulers and evaluate their sensitivity to the underlying system's level of heterogeneity.

This work has partly been published in the 10^{th} International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM 2019, held in conjunction with PPoPP 2019). Other aspects of the same project have been published in the proceedings of the International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS 2019).

1.1.3 Scheduler Selection

Generating a performance prediction matrix as described earlier, is only the first step of solving the scheduling problem for heterogeneous(-ISA) multicore architectures. Concluding to a schedule is equivalent to the mathematical problem of finding a row permutation of this prediction matrix, such that the sum on its diagonal is maximized. The diagonal of this matrix corresponds to the one-to-one job allocation, since its rows correspond to workloads and its columns to cores.

The "Hungarian" algorithm [51] solves this problem and guarantees finding (one of) the optimal schedules (row permutations) in polynomial algorithmic complexity. Consequently, the existence of such a scheduler is often assumed in proposals on performance prediction in the literature. We find however, that this algorithm does not scale well to larger chips with higher number of cores. Alternatively, many heuristics exist that exhibit lower algorithmic complexities, but do not guarantee an optimal solution.

Chapter 4 presents "Agon": a competitive scheduling approach, designed to scale to large heterogeneous (and heterogeneous-ISA) multicore systems. We find that the input to the scheduling problem (the particular selection of cores and workloads) can shift the scales in favor of lower-overhead, lower-performance schedulers. Furthermore, performance prediction error can render a scheduler incapable of finding a good schedule, regardless of its complexity and level of detail. Finally, as the size of heterogeneous multicore systems increases, the highest performing schedulers will not be a viable option due to their scalability issues, regardless of the added performance they can potentially offer. In this chapter, we train an artificially intelligent, competitive scheduler that selects a scheduling approach, depending on the current state of the system. Choices range from the most accurate hungarian algorithm, all the way down to a random scheduler. Agon is capable of choosing the optimal scheduler with accuracy up to 93%. Consequently, the system's performance improves compared to a system equipped with only a hungarian scheduler (the best scheduler on average). Performance of systems equipped with

Agon is within 1% of the performance of a system equipped with an oracular scheduler arbiter.

Joined with our work on high-quality performance predictions, our scheduler selector completes our proposal on scheduling for heterogeneous-ISA multicore architectures. This work is being prepared to be submitted for publication at the time of writing this dissertation.

Chapter 2

MemPod: A Clustered Architecture for Efficient and Scalable Migration in Flat Address Space Multi-level Memories

This Chapter introduces MemPod, a dynamic memory manager for flat address space memory configurations that is area efficient and high performance. It scales particularly well to future technologies with higher memory technology performance differentials.

In the heart of MemPod lies the *"Majority Element Algorithm" (MEA)* [55, 22]: A low overhead, high accuracy predictive algorithm, originally proposed for big data management and other database related applications, adapted to the needs and challenges of memory activity tracking and prediction. Studying this algorithm, reveals an interesting failure mode: Since memory activity can in no way guarantee the assumptions of the algorithm, we know beforehand that the algorithm would "fail". In failing however, it favors temporal locality - those memory accesses that happened most recently. This likely unintended feature, since it has no effect on the originally planned uses of the proposed algorithm, is extremely beneficial in the context of memory activity monitoring.

MemPod's novel microarchitectural design, clusters existing memory controllers into memory "Pods". Each Pod operates independently and in parallel, allowing for better scalability and integration to future systems with larger and faster memories and possibly a higher number of channels.

Throughout the course of this Chapter, we demonstrate how the lightweight and accurate predictive MEA unit, combined with MemPod's microarchitectural optimizations, leads to significant performance improvement.

2.1 Background

Die-stacked memory will increasingly be part of future systems, attempting to alleviate the memory wall [114]. Memory standards have been developed [47, 49, 84] and processor manufacturers are already announcing products featuring 3D-stacked memory [46, 82, 14, 7]. At the time of publication, this technology was limited to 8GB per stack [50] which does not fully address the capacity demands of modern systems. Therefore, die-stacked memories are expected to coexist with larger, slower off-chip memories, such as DDR4 or emerging byte addressable NVRAM [73, 91, 86] technologies, in a configuration often referred to as "Two-Level Memory" (TLM) [27, 72].

Stacked memory can be used as a large, high-bandwidth last level cache, or as part of main memory in a "flat address space". When used as a cache, recent research [27, 90] demonstrates the need to re-evaluate traditional SRAM-based cache organizations. Tag placement and granularity need to be re-evaluated, and we should avoid double memory accesses for tag check and data retrieval. Managing stacked memory as a cache is transparent to the software and improves the performance of latency-sensitive applications. However, capacity-sensitive applications do not gain significant improvements as the stacked memory's (typically significant) capacity is not utilized for additional storage.

In a flat address space configuration, the capacity of stacked memory can be allocated and used by applications. Dynamic memory managers proposed in the literature [96, 72, 27] monitor and profile memory accesses and attempt to transparently migrate frequently accessed pages to the fast portion of memory. While exposing more memory to the system, profiling memory accesses and performing transparent migrations often come with power, performance, and space overheads.

Software migration schemes [72] have high performance overheads and operate at coarse intervals, and thus are slow to adapt to changes in application phases. Recently proposed hardware managed schemes [96, 27] operate at finer granularity than software by either using simple, cache-like demand-driven migration or using a centralized management scheme. The former does not consider "hotness" of data, whereas the latter will not scale to large memories due to its centralized approach.

2.2 Related Work

A wide range of research proposals have sought to address the memory wall. Techniques such as *Bump[110], RMM[54] and Superpages[33]* attempt to optimize page placement in memory to expose higher parallelism. However, these scheduling mechanisms do not take advantage of a faster memory in a hybrid configuration.

Stacking DRAM dies in the processor package has been shown to achieve significant performance improvement. This technology cannot yet deliver large capacities [50]. Consequently, configurations combining stacked and off-chip memories have been proposed [69, 90] and can be found in the literature as "hybrid memories" or "two-level memories". The systems have proposed the use of the stacked memory either as a large high-bandwidth last level cache or as a "flat address space", where the capacity of the stacked memory is exposed to the software.

Organizing stacked memory as a cache has been explored in several studies [90, 28, 38, 77, 65, 48]. These approaches implement intelligent tag stores to allow cache-like operation while mitigating the cost of reading tags in DRAM. It has been demonstrated that traditional SRAM-tailored cache optimizations result in degraded performance when used in a DRAM cache and as such we need to "de-optimize for performance" [90]. DRAM cache organizations have been shown to improve performance significantly in latency-limited applications, while offering only marginal improvement with capacity-limited applications. It's been shown that exposing the extra capacity to the application instead of using it as a cache can benefit capacity-limited applications. To this end, recent work [72, 96, 27] proposes mechanisms to manage stacked memory as a flat address space.

HMA [72] is a HW/SW mechanism that attempts to predict frequently accessed pages in memory and, at predefined intervals, migrate those pages to fast memory. HW support is required for profiling memory accesses using counters for each memory page, while the migration is handled by the OS. Due to the costly OS involvement, HMA's intervals are kept large. Additionally, the hardware cost of its profiling counters is high. However, HMA is capable of managing migrations in a flat address space without the need of additional bookkeeping for finding migrated pages as the OS can update page tables and TLBs to reflect migrations.

Sim, et al. proposed a technique for transparent hardware management of a hybrid memory system [96], which we will refer to as "THM". THM does not require OS intervention while managing migrations. In order to keep bookkeeping costs manageable, THM allows migrations only within sets of pages (called segments). Each segment includes one fast memory page and a set of slow memory pages. The slow pages of each segment can only migrate to the one fast page location, and any such migration results in the eviction of the currently-residing page. THM monitors memory accesses with one "competing counter" per segment resulting in a low cost profiling solution. Finally, THM supports caching part of its structures on chip while the rest is stored in memory.

CAMEO [27] proposes a cache-like flat address space memory management scheme in an attempt to close the gap between cache and flat memory organizations. CAMEO operates similarly to THM, however it does so at the granularity of cache lines (64B). Migrations are restricted within segments with one fast line location per segment. Its bookkeeping structures are entirely stored in memory, while a "Line Location Predictor" attempts to save some bookkeeping-related accesses by predicting the location of a line. CAMEO initiates a line migration upon every access to slow memory.

Both THM and CAMEO sacrifice migration flexibility for area efficiency by restricting migrations in segments: if more than one hot page/line exists within the same segment only one can reside in fast memory. If no hot pages exist in a segment, its fast page cannot be utilized by another segment. Further, THM's competing counters can lead to false positives, allowing a cold page to migrate to fast memory, while CAMEO can incur high migration traffic as every access could induce a migration.

Spatial locality of applications can affect performance negatively when THM or CAMEO are used. Continuous pages or lines that lie within the same segment of each mechanism can be accessed frequently. THM is less susceptible to such issues because of its coarser granularity and the use of competing counters that will prevent a "ping-pong" effect. CAMEO, however is significantly affected. This issue is further exacerbated when the ratio between slow and fast memory capacities is increased. In such scenarios, under a configuration with 1:8 fast:slow memory ratio, both mechanisms suffer from reduced migration flexibility, e.g. forcing 8 slow pages to fight over a single fast page or line. In CAMEO's case, since every access to a slow line triggers a migration, a high slow-fast capacity ratio can result in the majority of accesses going to slow lines, causing a migration in every case.

2.3 Predicting Hot Regions With MEA

Migration mechanisms predict *future* hot pages to migrate them into fast memory. Prediction accuracy is critical to high performance, as each migration must be amortized by many future accesses to justify the cost of migration. A commonly used practice is to identify the hot regions within an interval and assume that those regions will be hot in the next interval. To accurately identify the hottest regions some mechanisms use an access counter per region. At the end of each interval the counters are sorted to identify the highest ranked (i.e., most accessed) regions. However, application phase changes could render this approach unsuccessful. Additionally, the number of necessary counters increases linearly as memory capacities grow.

To address the above limitations, we adopt a technique based on the Majority Element

Algorithm 1: Majority Element Algorithm

```
Input: X: Set of N elements
Input: K: Number of elements to output
Data: T: Map structure with K entries
Result: Set of K majority elements

Initialization: $T \leftarrow \emptyset$

foreach $i \in X$ do
    if $i \in T$ then
        $T[i] \leftarrow T[i] + 1$;
    else if $|T| < K - 1$ then
        $T[i] = 1$;
    else
        forall $j \in T$ do
            $T[j] \leftarrow T[j] - 1$;
            if $T[j] == 0$ then $T \leftarrow T \setminus j$;
        end
    end
end
```

Algorithm (MEA). MEA was originally proposed by Karp et al. [55] and was studied in-depth by Charikar et al. [22] for database management and big data analytics. This heuristic has formally been proven to correctly identify the K most frequently occuring elements of a set, when each of those elements appears more than $\frac{N}{K+1}$ (i.e. has majority), where N is the number of elements in the input set.

MEA is presented in Algorithm 1 as applied to an array of integers X. A map structure T maps K element IDs (in our integer array example, IDs are the integers' values) to K counters. Looping through the array, if the next integer exists in the map, its counter is incremented by 1. Otherwise, if there's enough room in the map a new entry is added with a count of 1. If the number does not exist in the map and all K counters are occupied, MEA subtracts 1 from every counter, removes the entries with a counter value of 0 and proceeds to the next integer. Once the entire array is processed, the map entries hold the majority elements.

A hardware implementation of MEA requires a mapping from page IDs to counters, as well as the area overhead of the counters themselves. On each access, the algorithm will perform one of the following operations: (a) find and increase exactly one counter by one, (b) add a new entry and set its counter value to 1 or (c) subtract one from all counters regardless of the page ID field and identify all the entries with a counter value of zero. All possible operations are simple and can complete within a single cycle if designed properly, using parallel subtractions and comparisons for operation (c).

In our application of MEA to activity tracking, the sequence of page addresses accessed correspond to the array of integers in the above example. However, we find that the sequence of accessed pages typically does not meet the condition of MEA that guarantees it will find the most-accessed pages; thus, it becomes an approximation. What makes MEA most useful, though, is its failure mode – when it fails to find the most-accessed pages, it does so by favoring recency over quantity. That is, a page accessed several times near the end of an interval can easily knock out a page accessed many more times early in the interval. As a result, it combines both access counting and temporal locality, at a fraction of the cost of access counting alone.

MEA's area overhead grows slowly with the amount of memory per pod. The number of counters can be kept constant, but the size of the ID or tag grows with the log of the memory size. Its $O(N)$ complexity works well for analyzing a stream of access requests in real time, and eliminates the need for sorting the counters.

In this section, we seek to understand the effectiveness of MEA's counting and prediction accuracy, compared against a Full Counters (FC) scheme, independent of the MemPod architecture. We use memory traces captured from multi-programmed 8-core workloads (the same traces used and described in Section 2.6) and simulate MEA and FC side-by-side with an in-house off-line simulator that provides oracle knowledge of future intervals.

The interval size for both MEA and FC was set at 5500 requests which is the average number of requests serviced within a 50us window in our timing experiments. For this experiment we use 128 MEA counters and FC requires 4.5M counters (assuming a 1+8GB memory capacity).

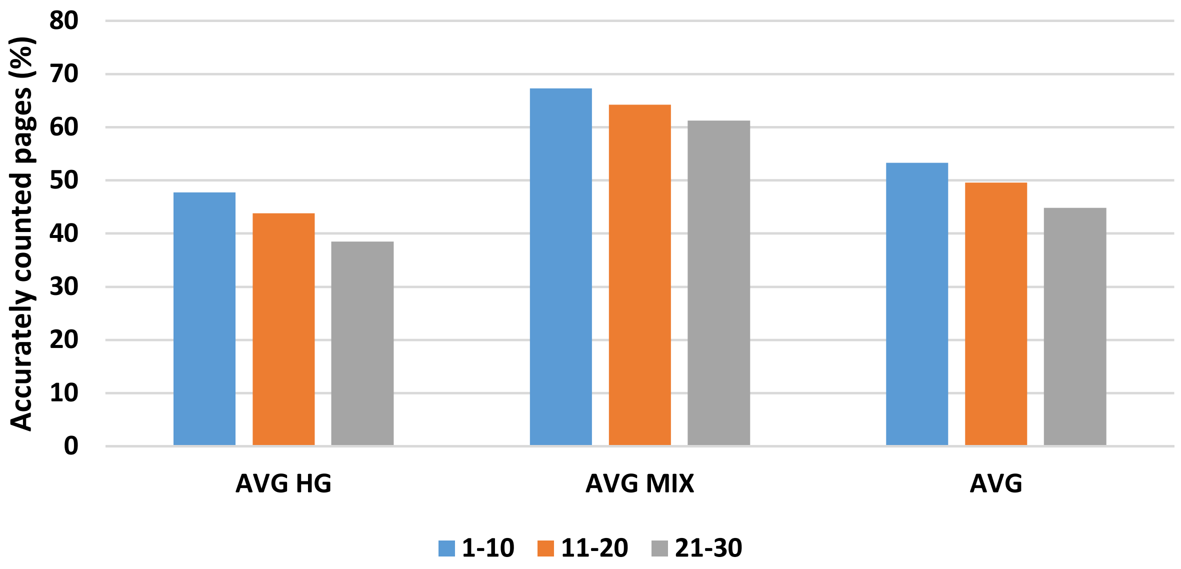


Figure 2.1. MEA counting accuracy compared to Full Counters on the top three tiers (ranks 1-10, 11-20, 21-30). Average results for homogeneous(AVG HG), mixed (AVG MIX) and all (AVG ALL) workloads shown.

We do two comparisons in this study. First, we examine the ability of MEA to identify the top pages in the past interval, something the full counter scheme will do perfectly. Second, we examine the ability of both schemes to predict the top pages in the next interval. In both studies, we will examine the schemes' ability to identify the 30 most-accessed pages, in bins of 10 each (1-10, 11-20, and 21-30).

Figure 2.1 shows the counting accuracy of MEA for the past interval, which should be compared with FC's perfect accuracy. In some workloads, MEA identified up to 75% of the top pages. However, on average MEA reports accuracy below 55% on the top tiers. Thus, it is a surprisingly ineffective replacement for accurate counters, if accurate counting were our priority. The bias toward recent accesses has a strong effect on the final value of the MEA counters.

When we instead examine the effectiveness in identifying future hot pages, we see a different story. Figure 2.2 presents a comparison of MEA and FC in terms of prediction accuracy. We compare each mechanism's "predictions" against the top three page tiers of the following interval based on oracular knowledge. Using 128 counters, MEA will return *up to* 128 predictions based on the past interval, while FC will return an overall ranking of each page accessed. In order to be able to directly compare accuracy, we take the top N pages from the full counters each interval, where N is the number of pages MEA returned.

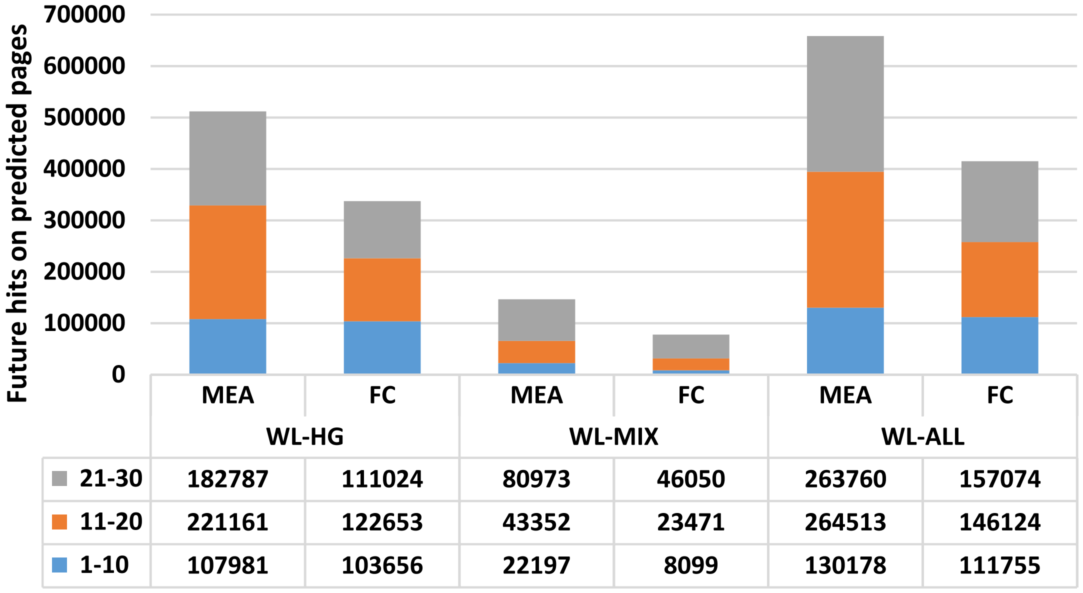
Figure 2.2 plots the number of hits on predicted hot pages from the previous interval. We also select interesting individual benchmarks and show them in Figure 2.3 to provide a more detailed comparison.

On average, MEA achieves more future hits than FC by 16%, 81% and 68% on the top three tiers respectively. Figure 2.3 shows selected individual workloads that generated interesting results and provides a more detailed comparison. Cactus[1] is the only workload where FC outperformed MEA's prediction. In fact it outperformed MEA on every tier.

Xalanc and mix9 are most representative of our overall results. We can see MEA outperforming FC's prediction accuracy in every bin. The last two workloads we selected, bwaves and lbm, show FC failing entirely to predict the future (FC also scored zero future hits with our libquantum workload). With bwaves (and libquantum) MEA reports a very low number of future hits but not zero. These results can happen when an application streams through large structures that exceed the size of the interval. In that case, the past interval has little overlap with the next interval, but recent accesses are much more likely to be overlapped. Lbm shows an interesting result, where MEA reports a high number of hits (outside of the first tier) in a workload where FC failed entirely. This can happen with a large working set where the application does a fairly constant amount of work per page. Full counters, then, will record the highest access counts for pages the application is done with, while MEA will favor pages the application was still working on at the end of the interval.

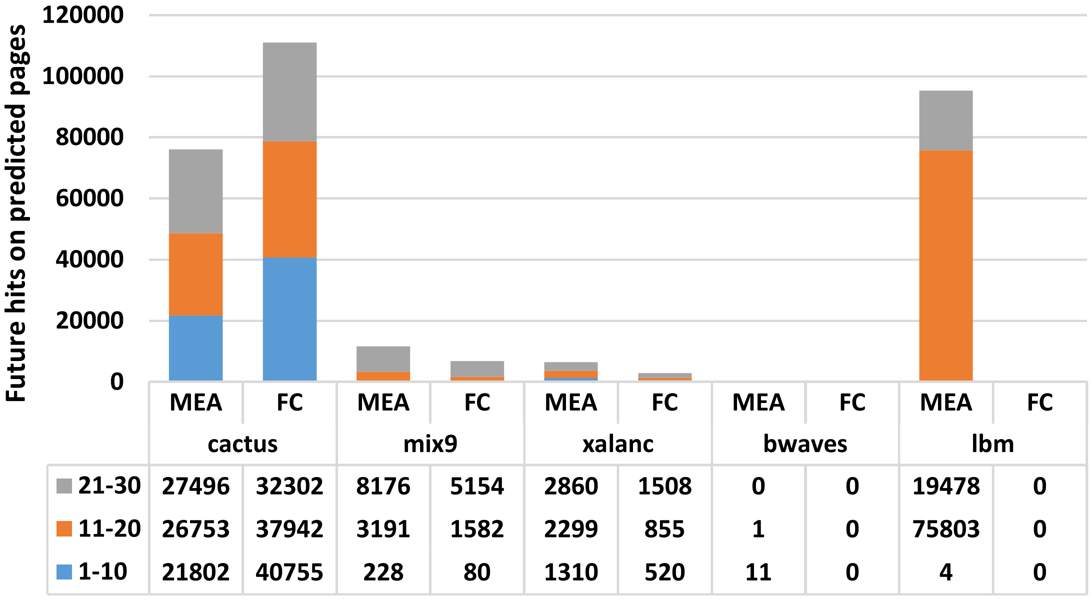
MEA has not previously been used in any kind of architectural event tracking; however, our results indicate it is an attractive alternative to full counters at a very small fraction of the hardware cost.

[1]We use a single benchmark's name as a shorthand for workloads running the same benchmark 8 times simultaneously on 8 cores.



	WL-HG		WL-MIX		WL-ALL	
	MEA	FC	MEA	FC	MEA	FC
■ 21-30	182787	111024	80973	46050	263760	157074
■ 11-20	221161	122653	43352	23471	264513	146124
■ 1-10	107981	103656	22197	8099	130178	111755

Figure 2.2. MEA prediction accuracy (part 1) compared to Full Counters on the top three tiers (ranks 1-10, 11-20, 21-30). Results for homogeneous(WL-HG), mixed (WL-MIX) and all (WL-ALL) workloads shown.



	cactus		mix9		xalanc		bwaves		lbm	
	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC
■ 21-30	27496	32302	8176	5154	2860	1508	0	0	19478	0
■ 11-20	26753	37942	3191	1582	2299	855	1	0	75803	0
■ 1-10	21802	40755	228	80	1310	520	11	0	4	0

Figure 2.3. MEA prediction accuracy (part 2). This graph presents the most interesting results from individual workloads.

2.4 Migration Building Blocks

The design of a complete memory manager can be broken down into the following 5 “building blocks”:

Migration flexibility: This is defined by the possible mappings in fast memory that a particular memory region (page) can map to.

Remap table: A structure that keeps track of migrated pages and is able to provide a relay address given a requested address.

Activity tracking: Logic and structures needed to profile memory requests and predict future “hot” pages.

Migration trigger: Defines when migration occurs. Commonly the trigger can be event, interval, or threshold-based.

Migration driver/datapath: Defines the path followed and the hardware modules involved in performing migrations.

Each of the above building blocks introduces some trade-off. For example, allowing more flexibility in migration locations can lead to higher performance benefits, at the cost of larger book-keeping structures. The design choices for the various building blocks are largely orthogonal. Thus, architects can select an approach to each building block suitable to their system’s capabilities and limitations and simply combine them to create a desirable memory management mechanism.

2.4.1 Migration Flexibility and Remap Table Size

Migrating pages can provide the highest benefit when no restrictions are imposed on the available migration locations. This amount of flexibility, however, requires more bookkeeping and can incur a higher cost.

A hardware-driven migration mechanism requires some kind of remap table, commonly implemented as a hash structure, indexed by a page's address and pointing to the migrated (or relay) page address if one exists. On a page migration, the remap table is updated to reflect the new address of a migrated page.

The remap table should provide the remap (relay) address for each original page address. Some other structures may also be necessary to avoid expensive table searches when an inverse lookup is needed (e.g., to identify pages currently mapped to slow or fast memory).

2.4.2 Activity Tracking

Activity tracking is a critical element of any management mechanism for hybrid memories. In most studies on the subject, activity tracking becomes a synonym for identifying hot regions by counting the number of accesses to each one. In a more generalized approach, it could potentially be extended to track patterns, parallelism, bit flips/faults or any other information useful to the underlying mechanism.

The overhead of maintaining a set of counters per memory page (or other granularity) will be high. Space requirements will increase linearly as memory capacities grow and the cost of sorting all the counters can overshadow any potential benefits in performance. Furthermore, our evaluation presented in Section 2.3 demonstrates that using full counters to ensure 100% accurate counting may still lead to poor prediction accuracy. Frequently encountered cost-reducing solutions in the literature consist of increasing the activity tracking granularity in order to reduce the number of counters needed (i.e. track a group of pages together), limiting the bit width of counters, and caching a subset of the tracking state while the full set resides in main memory.

2.4.3 Migration Triggers

Each memory management policy must decide when to trigger migrations. Migrations add significant delays to a system and any penalties incurred should be amortized by the performance improvement from placing a page in the fast memory. Requests that arrive while migrations are being performed have to be delayed to ensure functionally correct memory behavior. Throughout the literature, three triggers are most commonly used whenever state must be updated based on tracking information (MC scheduling, migrations, dynamic voltage and frequency scaling etc.). Interval-based (or epoch-based) triggers occur with a set frequency, while threshold-based solutions trigger whenever a predetermined criterion is met. Finally, event-based triggers react to predefined events. Both interval-based and threshold-based approaches face the same challenge of identifying the optimal interval or threshold value.

2.4.4 Migration Datapath

Regardless of the choice for each migration building block described so far, once migration is triggered, the migration manager has to follow a number of steps: First, migration candidates need to be identified. Traditionally, one page (or a segment/line depending on the migration granularity) from the slow memory and one from fast memory. Then, the two identified candidates need to be swapped. During the swap process, one or both pages will be read and stored in temporary buffers and then written at their remapped locations.

Without dedicated migration driver hardware, migrations will have to be orchestrated by the OS and CPU cores. Consequences include communication delay, potentially some pollution of the processors' caches and the unavailability of those CPUs during migration.

MemPod implements the migration driver within each Pod. As each Pod has direct communication with its member MCs, added delays are kept to a minimum, and no traffic is generated at the global switch (saving energy and eliminating contention). In HMA, the OS orchestrates everything. Some CPU cores have to be stalled and used to service the OS interrupt, causing the migrated pages to traverse through communication mediums and caches on each way. THM does not fully describe its datapath, but it appears that CPUs are used in this case as well. CAMEO describes its swapping operation to be transparent to the OS by using existing writeback and fill queues. Its mechanism relies on the two memories sending writeback or demand read requests to each other, which further implies some added logic in Memory Controllers, as well as communication between MCs.

To make a generous comparison, we do not model the penalty introduced by using CPUs or global communication mediums for migration in our HMA, THM, and CAMEO simulations presented in Section 2.6.

2.5 Architecture

Our clustered migration mechanism is designed to address key challenges associated with the migration problem. In this section, we first present a high-level overview of MemPod's proposed micro-architectural design, followed by a more detailed discussion regarding each building block. Throughout this section we also discuss some of the major design decisions of the state-of-the-art mechanisms.

2.5.1 Clustered Migration Architecture

Figure 2.4 presents an overview of MemPod. MemPod's design was kept modular to facilitate system integration and scalability. A number of memory "Pods" are injected between the Last Level Cache (LLC) and the system's memory controllers (MCs). Each Pod clusters a number of MCs and enforces migrations to only occur among its member MCs. Pods do not communicate with each other, preventing inter-Pod migrations. To the rest of the system, Pods are exposed as MCs. With MemPod's transparent design, each Pod will now be receiving all the requests originally addressed to any of the Pod's member MCs.

When a memory request arrives, the Pod monitors the request, updates any necessary migration-related activity tracking counters, and forwards the request to the intended recipient MC. The migration logic within a Pod does not need to be invoked during a response from any MC and can be bypassed to reduce memory access latency. A drawback of clustering MCs into Pods is the serialization of potentially parallel requests to different MCs of a single Pod. As such, activity tracking within a Pod as well as the subsequent forwarding of requests must be as efficient as possible.

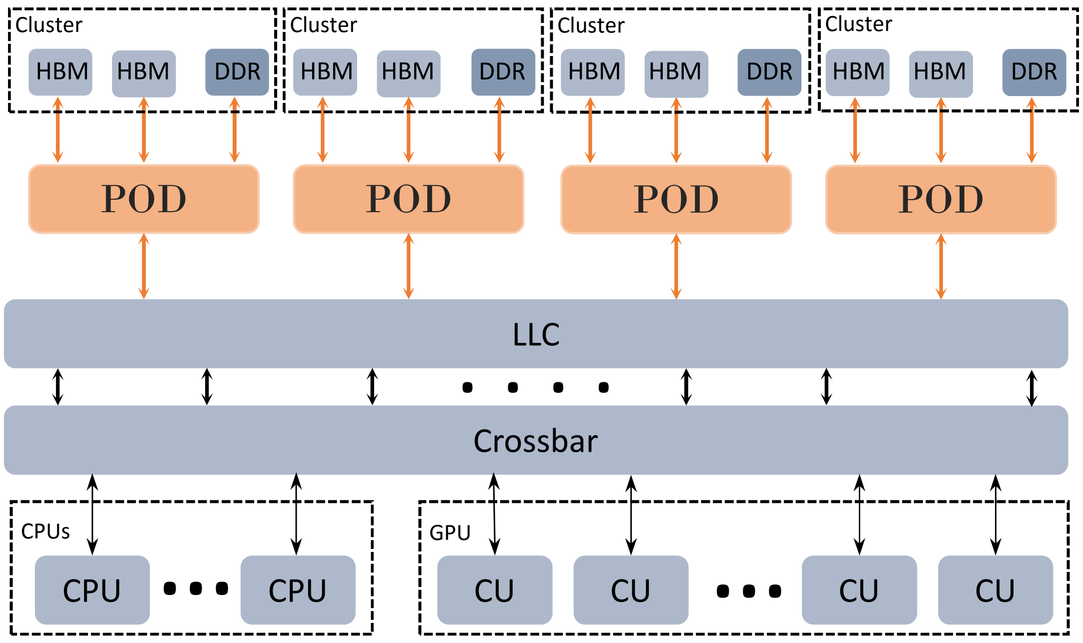


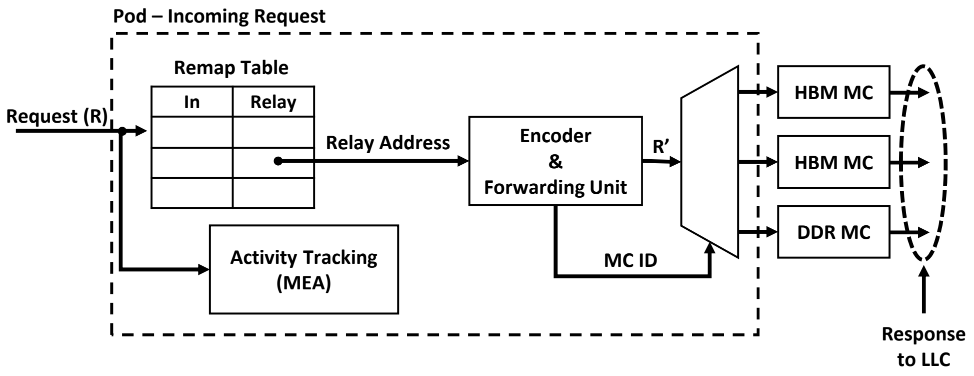
Figure 2.4. MemPod high-level architecture

MemPod's clustered architecture also reduces global traffic during migrations compared to non-partitioned mechanisms. Because migration traffic happens within a Pod, this architecture significantly reduces global traffic and enables parallel migrations.

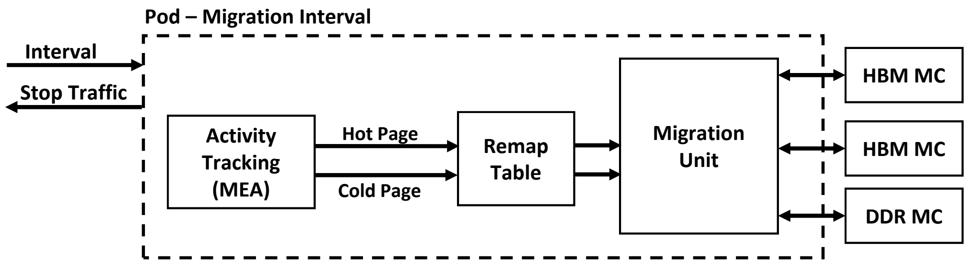
Memory Pod

The major architectural elements of a Pod are shown in Figure 2.5. A Pod includes an activity tracking (MEA) unit, a remap table for keeping track of migrated pages and a forwarding unit that can re-encode a request with the relay address and, based on that address, send the request to the appropriate MC.

A designer can vary the parallelism and flexibility of MemPod by varying the number of Pods. A design with one Pod is equivalent to a centralized migration controller allowing any-to-any migration, while a design with a Pod number equal to the number of MCs would imply that migration is disabled. A reasonable design point would be to set the number of Pods equal to the number of slow-memory MCs. Such a configuration inherently prevents migration between slow off-chip channels, while at the same time maintaining full channel-level parallelism



(a) Pod's request forwarding operation



(b) Pod's migration procedure

Figure 2.5. Major architectural Pod elements

on the system's bottleneck: the slow MCs. In a configuration where the number of fast-memory MCs are not a multiple of slow-memory MCs, Pods can be configured asymmetrically or some MCs could be members of multiple Pods, with their capacity partitioned to avoid crosstalk issues. In Figure 2.4 we present a system with eight MCs for the fast, on-die stacked memory and four MCs for the slow off-chip memory. Throughout this paper we use die-stacked HBM as the fast memory [50] and DDR4-1600 as our off-chip memory, and we set the number of Pods to four, as shown in Figure 2.4.

2.5.2 Building Blocks

MemPod imposes few migration restrictions since each slow page can migrate to any fast page location as long as it's within the same Pod. To support high flexibility MemPod requires a Remap Table structure capable of tracking all pages at each Pod and upon a lookup return the new address. The page table is updated to reflect changes with each migration. In addition to the

Table 2.1. Breakdown of state-of-the-art designs

Challenge	**Tradeoff**	**THM**	**HMA**	**CAMEO**	**MemPod**
Page Relocation	*Flexibility / Time*	Only 1 Candidate	No Restrictions	Only 1 Candidate	Intra-Pod Migration
Remap Table Size	*Flexibility / Area*	1 entry per fast page (1.5kB)	No remap table	1 entry per fast line (72kB)	1 entry per page (2.8 MB / Pod)
Activity Tracking	*Accuracy / Area*	8 bits per fast page (512kB)	16 bits per page (9MB)	N/A	64 MEA entries (736 B)
Migration Trigger	*N/A*	Threshold	Interval	Event	Interval
Tracking Organization	*Simplicity / Parallelization*	Fully centralized (Serialization)	Fully distributed	Fully distributed	Semi-distributed (Pods)
Migration Driver	*Latency*	CPU	CPU (OS)	MCs	Pod
Migration Cost	*Time*	HW cost + CPU	HW cost + SW + TLB + CPU	HW Cost + Communication	HW

remap table, our algorithm also needs to identify all pages currently mapped to fast memory (to identify a candidate to be evicted in favor of a new hot page). We do this with a smaller, inverted table that gives the original address of each page currently mapped to fast memory.

MemPod requires an MEA map structure of K entries, where K is the number of hot pages we wish to identify at each interval. Our evaluation presented in Section 2.6 finds a good number of MEA counters to be 64. Each entry maps a page's address to a counter. Through our evaluation, we identified a good counter size to be 2 bits and 21 bits are needed to address each page within a Pod, leading to a total storage cost of 736B. Using the MEA counters, MemPod's activity tracking profiles *every page in memory* with minimal hardware cost.

MemPod uses timing intervals. At each interval a Pod will migrate up to K pages into its fast memory, where K is the number of MEA counters used. MemPod is transparent to the system, rendering costly OS intervention unnecessary. Since each one of the N Pods will attempt to migrate up to K pages, up to N$\times$K migrations can happen within each interval. However, all Pods can perform their migrations in parallel. Due to MemPod's lightweight activity tracking, intervals can be kept very small, allowing each Pod to better adapt to the application's phase changes. Our evaluation shows a good interval length to be 50us.

With the use of MEA counters, identifying the fast-memory page candidate is as simple as checking that it's not part of the K hot pages. The identification algorithm starts at the very first fast memory location and iterates sequentially until it detects a page address that is not in the set of hottest pages. For the next migration, the identification algorithm simply continues from where it left off. If a hot page already resides in the fast memory it is ignored.

In the state of the art mechanisms presented and evaluated in this paper, building block decisions vary significantly. HMA does not require a remap table due to the OS updating the existing system's structures. For activity tracking it uses Full Counters. The costly OS involvement and the high penalty for sorting all its counters force HMA to operate at very large intervals, weakening its adaptability to phase changes. However, HMA offers full flexibility for migrations.

THM offers significantly limited flexibility by restricting migrations withing segments, however this decision reduces bookkeeping costs significantly. Competing counters in each segment are used for activity tracking, occasionally leading to false (threshold-based) migration triggering if a cold page gets accessed at the right time. Identifying migration candidates incurs very little overhead since there is exactly one fast memory location for each slow memory page that triggers migration.

CAMEO operates similarly to THM, restricting migrations within segments (called congruence groups) but it operates at a finer granularity. Due to its finer granularity, it requires a larger remap table structure than THM. CAMEO does not require activity tracking since it uses an event based migration trigger performing a swap at each slow memory access. The process for identifying migration candidates is identical to THM's.

Table 2.1 shows a detailed comparison of all mechanisms' building block decisions, comparing their costs and presenting the tradeoff impact of each one.

2.5.3 Distributed Migration Controllers

Migration mechanisms in the literature [96, 27] assume a centralized migration controller which all memory requests have to go through before reaching the memory, in order to monitor memory activity, read remap tables and control migrations. MemPod's distributed migration controllers control every aspect of the migration process. The use of four Pods breaks the problem into smaller pieces in a divide-and-conquer approach. Instead of trying to identify the N best pages to migrate from the entire memory, each Pod now has to identify the $\frac{N}{4}$ pages per Pod. Each Pod will also require lower bandwidth than a centralized unit as it handles a fraction of the traffic.

Prior publications [68] demonstrate that the layout and address interleaving of main memory and last level cache can be co-designed and benefit a system by increasing efficiency and reducing global traffic. The proposed designs align cache "banks" with main memory banks (among other optimizations). Even though not specifically proposed for 3D-stacked memories, if such a design is assumed, a centralized migration controller would be detrimental to the carefully designed alignment since all LLC misses will now have to go through the centralized unit and then fan back out to their respective MCs.

Finally, a clustered design ensures that we are never moving data across the entire system. Migration can only occur within a Pod and between "sibling" MCs. By limiting migration distance, MemPod imposes a tighter ceiling on data movement energy which can lead to migration-related energy savings when compared to a centralized design.

2.6 Results

2.6.1 Evaluation Framework

The goal of our evaluation framework is to quantitatively and qualitatively assess MemPod's capabilities and compare it against state-of-the-art proposed mechanisms. Throughout our evaluation section, we study MemPod's performance running with an eight-core CPU. We extend Ramulator [57] to support flat address space hybrid memories. We model the MemPod architecture, as well as HMA, THM, and CAMEO in our simulation framework. Ramulator enables cycle-level memory simulation and includes a simple CPU front-end capable of approximating resource-induced stalls. We evaluate MemPod under a realistic memory configuration consisting of 1GB 3D-stacked HBM [50] and 8GB of off-chip DDR4-1600. Table 2.2 Provides a more detailed description of the simulated system's configuration.

2.6.2 Experimental Methodology

We use benchmarks from the SPEC2006 suite [41] as our workloads. Using Sniper [21], we record memory request traces while simultaneously executing 8 benchmarks on a simulated 8-core CPU. We then feed these multi-programmed memory traces into Ramulator, executing all workloads to completion. Our complete set of workloads consists of 15 "homogeneous" workloads, where 8 copies of the same benchmark are run in parallel (we simply refer to these

Table 2.2. Experimental framework configuration

Processor	
Cores	8 @ 3.2GHz
Pipeline	4 wide out-of-order

Caches	
L1 I-Cache(private)	64KB, 2 way, 4 cycles
L1 D-Cache (private)	16KB, 4 way, 4 cycles
L2 Cache (shared)	8MB, 16 way, 11 cycles

	HBM	DDR4-1600
Capacity	1GB	8GB
Bus Frequency	1 GHz	800 MHz
Bus Width (bits)	128	64
Channels	8	4
Ranks	1	1
Banks	16	16
Row Buffer Size	8KB	8KB
tCAS-tRCD-tRP-tRAS	7-7-7-17	11-11-11-28

workloads by the benchmark's name in later results), as well as 12 workloads featuring a random mix of 8 benchmarks each (referred to as mix1-12). Each benchmark is executed and traced under its reference input. When running homogeneous workloads, Sniper ensures that memory pages are not shared between workloads. A breakdown of the mixed workloads is shown in Table 2.3.

We also extended Ramulator with caching for the activity tracking and/or remap tables depending on the simulated mechanism. Bookkeeping-related cache misses inject memory requests into the stream of requests fed by our trace files to retrieve the missing information. No priority is given to these cache miss requests over regular requests. In experiments where the caches for hybrid memory management techniques are disabled, the simulator assumes that any information needed by any mechanism exists on chip and is accessible without any delay. The migration process was implemented in detail as well. In order to read an entire 2KB DRAM page from memory, 32 read requests need to be sent for each of the two migration candidates and then another set of 32 requests for each of the two write-backs.

As we use Ramulator with recorded traces, we report Average Main Memory Access Time (AMMAT) in our results. Even though Ramulator has the ability to approximate IPC with a

Table 2.3. Mixed workloads description

	mix1	mix2	mix3	mix4	mix5	mix6	mix7	mix8	mix9	mix10	mix11	mix12
astar		✓					✓	✓	✓✓		✓✓	
bwaves					✓		✓	✓✓	✓			✓
bzip				✓	✓	✓	✓	✓				✓✓
cactus				✓	✓✓	✓		✓				✓✓
dealII				✓✓	✓		✓	✓	✓			✓✓
gcc	✓	✓	✓	✓		✓✓				✓✓		
gems	✓	✓					✓		✓		✓	
lbm	✓		✓			✓				✓		
leslie	✓	✓	✓				✓		✓		✓✓	
libquantum			✓			✓				✓✓		
mcf	✓	✓	✓	✓✓	✓	✓				✓		
milc	✓		✓	✓						✓		
omnetpp	✓	✓						✓			✓	
soplex			✓		✓	✓				✓		
sphinx		✓	✓						✓		✓	
xalanc					✓		✓	✓				✓
zeusmp	✓	✓					✓		✓		✓	

simple CPU model, AMMAT is computed with much greater fidelity with this tool, as it models the memory system in great detail. AMMAT is the average time spent accessing and waiting for main memory by each request (lower is better).

In our AMMAT experiments, we typically introduce additional accesses to the system (migrations, bookkeeping cache misses). The overhead of the additional misses is accounted to the total memory stall time, but the total memory stall time is divided by the number of original LLC misses (main memory requests) captured in our traces – that is, the denominator in our AMMAT equation does not change between experiments and equals the number of requests in our trace file.

Due to space limitations we are not able to show results for individual workloads in most of the graphs in this paper. In those graphs, we only present the average of all mixed workloads, average of all homogeneous workloads, and overall average.

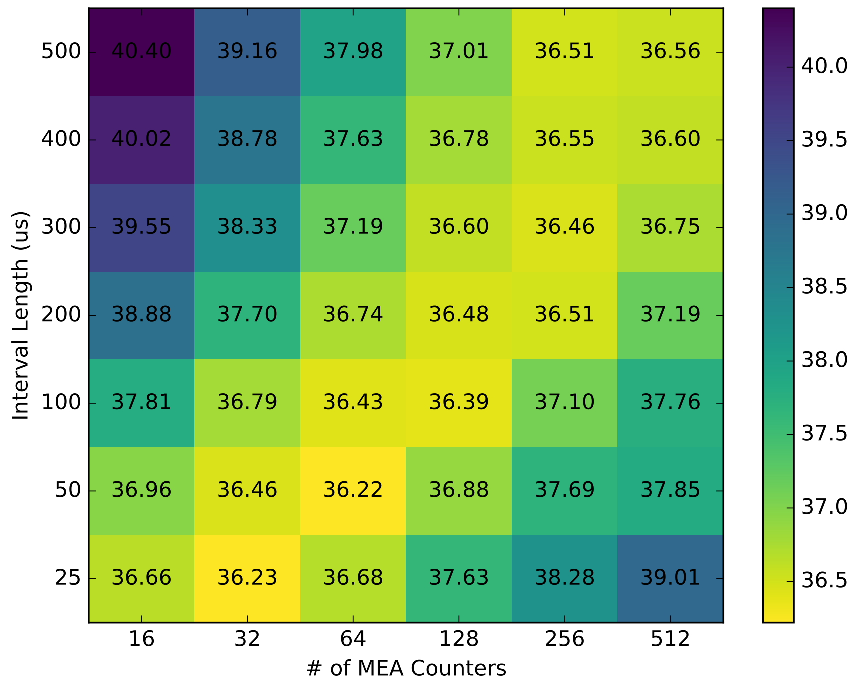


Figure 2.6. Average AMMAT from all workloads under various MemPod configurations.

2.6.3 Simulation Results

Page Tracking and Migration Design Space

MemPod's activity tracking overhead and migration traffic is impacted by the number and size of the MEA counters, as well as the epoch (interval) size over which the counters accumulate. We examine each of these design space parameters in this section. The number of MEA counters dictates the highest possible number of migrations that can be performed at each interval by each Pod, while the epoch length will determine MemPod's ability to better adapt to phase changes in a workload. Furthermore, a small number of counters favors less aggressive migration at each interval boundary. The size of each MEA counter sacrifices accuracy when smaller counters are used but can also save space on the chip. Smaller counters also sacrifice previous-interval counter accuracy for recency, as it makes it easier to replace a counter for a previously hot page no longer being accessed.

We first identify the optimal number of counters and epoch length by running a series of experiments. We executed all combinations of epoch length and number of counters pairs with epoch lengths of 25-500us. We also exponentially increase the number of MEA counters per pod from 16 to 512. In order to minimize the impact of other factors, we execute this experiment with 16 bits per counter and remap table caches disabled.

Figure 2.6 shows our results obtained by taking the average AMMAT from all workloads under each MemPod configuration. Based on the results we derive some observations:

- MemPod achieves the best performance (lower AMMAT) with 50us intervals and 64 counters per Pod. MemPod's lightweight operation allows for such small intervals. For comparison purposes, HMA [72] identified the best epoch length to be 100ms (2000x larger) in order to support all the lengthy processes that take place during a migration event for that method.
- The lowest AMMAT values lie on the diagonal of our result matrix. This implies that the key determinant is the number of migrations, as the maximum migration rate is (roughly) a constant across the diagonal.
- Few counters and long intervals (inability to react well to phase changes), at least in this sweep of parameters, appears to be worse than many counters and small intervals (overly aggressive migration activity).

The size (in bits) of each counter defines the area requirements of our MEA tracking mechanism.

Figure 2.7a presents AMMAT results normalized to a configuration with 2-bit counters as well as the average number of migrations per Pod per epoch (secondary axis). We first observe that 8 bits are sufficient for our workloads, as larger sizes report identical results. We also observe that reducing the counter size to even less than 8 bits benefits performance, although very marginally. Small counters, as mentioned, favors recency. The smaller the interval, the more important recency becomes over accurate counting; plus, the fewer bits required for accurate counting since there are fewer events. Finally, for 50 us intervals, 2 bit counters offer the best performance (but again, the differences are small).

Figure 2.7b shows the same experiment with MemPod's parameters set to 100us intervals and 128 counters. This figure demonstrates that as the interval and number of counters increases (i.e. less focus on temporal locality required), the optimal counter size grows from 2 to 4 bits.

Based on these results, we use 64 MEA 2-bit counters over 50us intervals for subsequent results in this paper. Each one of the 64 MEA entries needs 21 bits for addressing the 1.1M pages per Pod and 2 bits for its counter, leading to an area cost of only 184B per Pod and 736B total. Compared to the state of the art, MemPod's activity tracking requirement is $\sim$712x smaller than THM's (512KB) and $\sim$12800x smaller than HMA's (9MB).

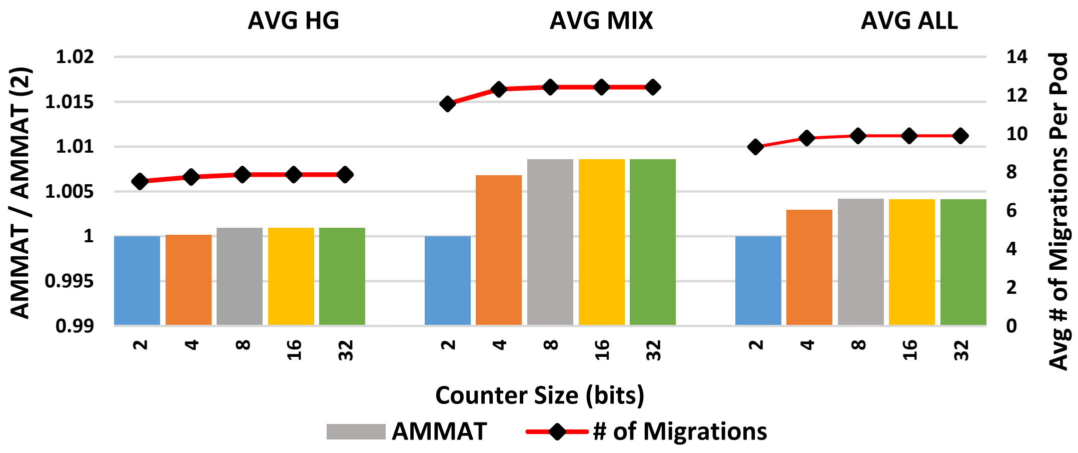
Performance Comparison

Figure 2.8 presents a performance comparison of MemPod, HMA, THM, CAMEO and a configuration with 9GBs of on-chip HBM memory, normalized to the performance of a hybrid memory configuration without migration capabilities. We evaluat all mechanisms with migration-related caching disabled.

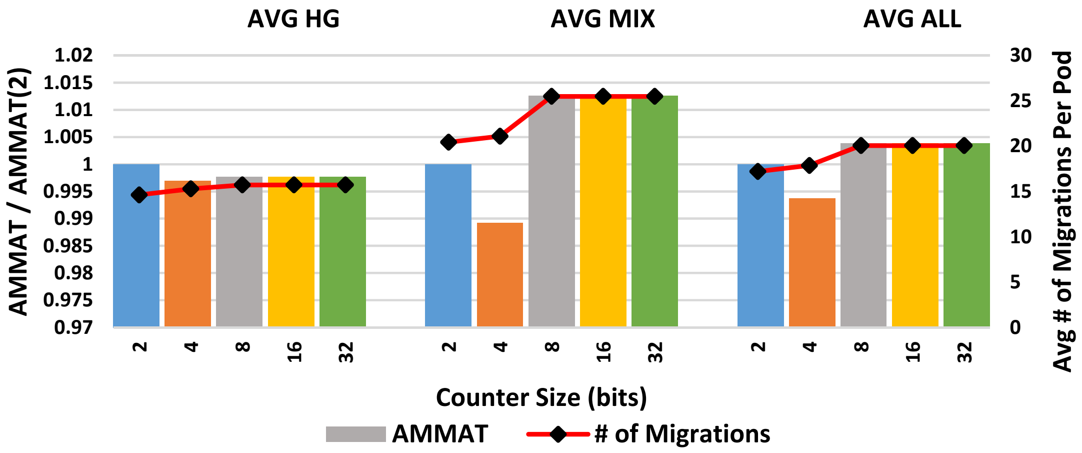
In order to model HMA's penalties more accurately, we profiled sorting 4.5M integers using quicksort (NlogN complexity) on a real system with a recent Intel Core i7 processor running at 2.1GHz. Our experiment showed that sorting all of HMA's activity tracking counters, at these memory sizes, takes 1.95s on average. When we scale this overhead to our simulated 3.2GHz core, the expected delay would be approximately 1.2s, which is much larger than the proposed optimal interval size for HMA. We instead assumed a very generously reduced overhead of 7ms per interval, assuming we could sort in parallel and discard obvious low values before sorting.

Based on the results we make the following observations:

- MemPod outperforms the state-of-the-art competitors in the majority of our workloads, and in several cases is very close to an HBM-only configuration. MemPod improves AMMAT over a two-level memory without migrations by 19% on average.
- On average, CAMEO reports AMMAT degradation of 41% over the no-migration scheme.



(a) MemPod @ 50us, 64 MEA counters



(b) MemPod @ 100us, 128 MEA counters

Figure 2.7. Counter size (in bits) Vs Normalized AMMAT (primary axis) and average # of Migrations per Pod per interval (secondary axis)

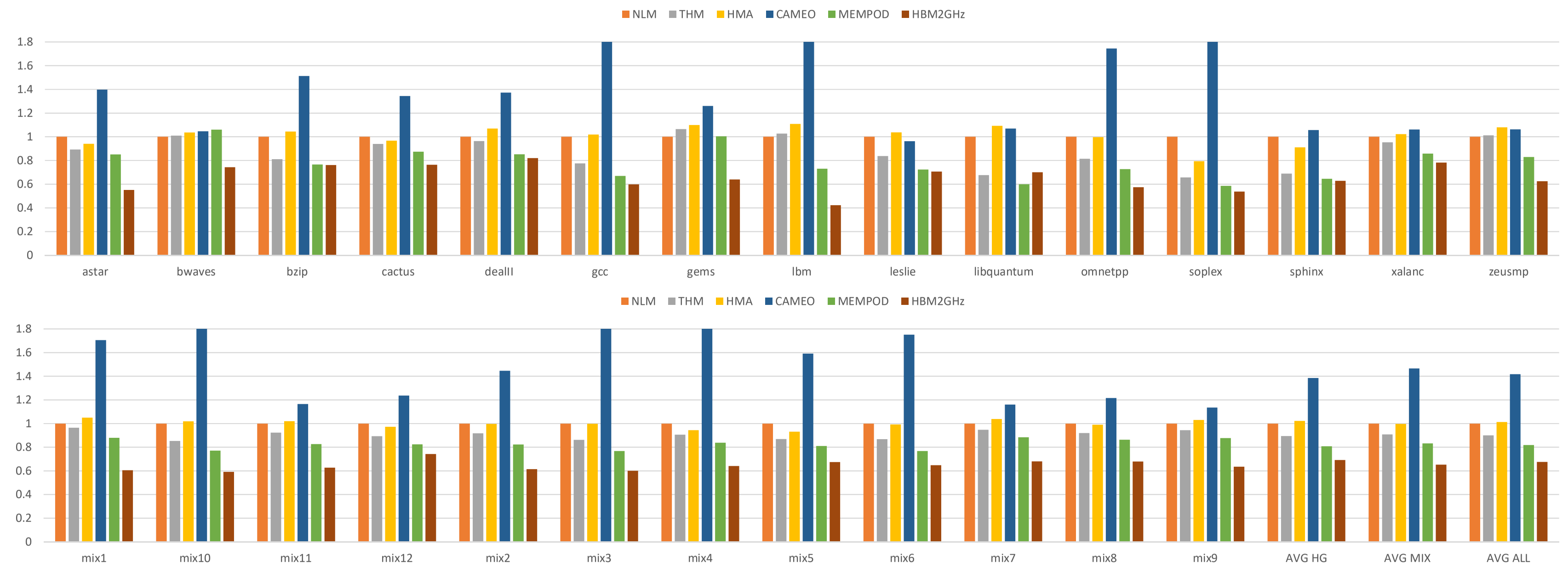


Figure 2.8. Performance Comparison: AMMAT is normalized to a hybrid memory without any migration mechanism.

The negative impact is caused by our high ratio of slow to fast memory capacity. Given CAMEO's algorithm, 9 slow lines compete for one fast line. At that ratio, it is much more likely for two or more lines to thrash competing for the one spot. From our experiments, we observe CAMEO to force the most movement despite the fact that each move is much smaller. CAMEO moves 3.9GB of data on average per 8-core experiment. For comparison purposes, MemPod moved 3.1GB on average, however migration traffic was divided between Pods, resulting in 804MB per Pod. THM moved 865MB on average and HMA moved 578MB due to its large intervals. We also frequently observe wasted migrations with CAMEO, where a line is evicted before it is touched.

- On average MemPod reports 21% higher AMMAT than HBM-only, while THM and HMA report 33% and 40% respectively.
- In some workloads migration overall is harmful to performance, as observed with the bwaves workload, where a no-migration scheme reports higher performance (lower AMMAT). We observe that in those cases, MemPod leads to deteriorated performance compared to the other mechanisms. However, in the cases of lbm and zeusmp, MemPod increases performance, while THM, HMA and CAMEO report higher AMMAT than the no-migration scheme.
- HMA and MemPod outperform HBM-only when executing the libquantum experiment. In the case of libquantum, the working set size fits entirely in our fast memory. As a result, after some migrations, the entire working set will be present in HBM. With an HBM-only system and no migrations, pages are inserted sequentially by address. In a migration-based system, simultaneously-hot pages are inserted together after each epoch. As the DRAM row buffer is bigger than a page, we find that the co-location of simultaneously-hot pages increases row buffer hit rate from 7% (HBM only) to 90% (MemPod), with 87% of those taking place in fast memory. HMA also sees an improvement in row buffer hit rate. THM and CAMEO cannot take advantage of the small footprint due to their restricted migration

flexibility (only one hot page/line per segment can reside in fast memory).

Caching Effect

Migration mechanisms will be forced to include a cache as activity tracking and remap table structures are too large to store on-chip. The use of a cache will unavoidably hinder performance. Each mechanism has different cache requirements. THM caches its counters and remap table together with its SRT structure. HMA has no need for a remap table but has high storage requirements for its counting mechanism. MemPod must cache only its remap table as MEA counters easily fit on chip.

In this experiment we evaluate the impact of a cache on MemPod's performance and we also compare it against HMA and THM when operating with a cache. MemPod was configured with the optimal parameter values identified over the course of this section. Every mechanism was evaluated with 16, 32 and 64 KB of cache. For MemPod, the cache capacity is distributed equally over its four Pods. A part of stacked memory was partitioned to serve as each mechanism's backing store.

In our implementation, each cache miss injects a read request to retrieve missing data. Since all of MemPod's cache misses will occur due to Remap Table updates or lookups it becomes a blocking request for the affected page. All incoming requests to that page need to be delayed until the missing data is retrieved. In-flight requests are not affected by incoming cache misses.

Figure 2.9 shows our results obtained by taking the average AMMAT from all our workloads for each cache size for each mechanism. We present AMMAT results normalized to a two-level memory configuration without any migration mechanism. With 16, 32 and 64kB of cache, MemPod reports 4, 7 and 9% AMMAT improvement over a 2-Level Memory (2LM) with no migration capability and outperforms the other mechanisms.

For 16, 32 and 64kB of cache, the impact on MemPod's performance (when compared to its performance without a cache) is 16, 14 and 12% respectively. THM reports impacts of 12, 10

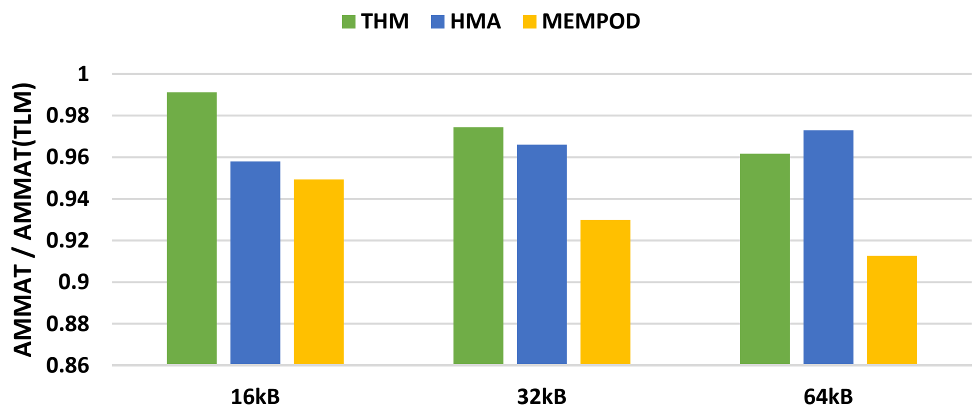


Figure 2.9. Sensitivity analysis: Cache size. AMMAT normalized to a TLM memory.

and 9%. Interestingly, HMA reports lower performance impact with smaller cache sizes rather than larger. Our investigation revealed that the extra cache misses caused by a smaller cache lead to a reduced number of incoming requests to be serviced per interval. As a result, HMA's activity tracking counters have lower values at the end of the interval, leading to fewer migrations. As demonstrated earlier in Section 2.3, HMA (using Full Counters) has a low prediction accuracy. As such, by reducing the number of migrations, we observe the number of requests serviced by the fast memory to increase, due to hot pages that were not replaced as aggressively.

Scalability

MemPod is designed to be scalable with future technology. If we grow memory sizes by increasing the number of pods, the size of the remap table and the size of the MEA counters will remain constant (per pod, and thus per memory page) if the memory per pod remains constant. If instead we increase memory capacity per pod, the size of the remap table (per memory page) will go up only with the log of the per-pod memory. If we choose to scale the number of counters with the size of memory per pod, it will go up similarly; however, if we do not scale up the counters at the same rate (e.g., four times the memory, but double the counters), then the cost of the counters (per memory page) will go down.

Additionally, the memory traffic caused by migrations will remain distributed and off the primary processor interconnect.

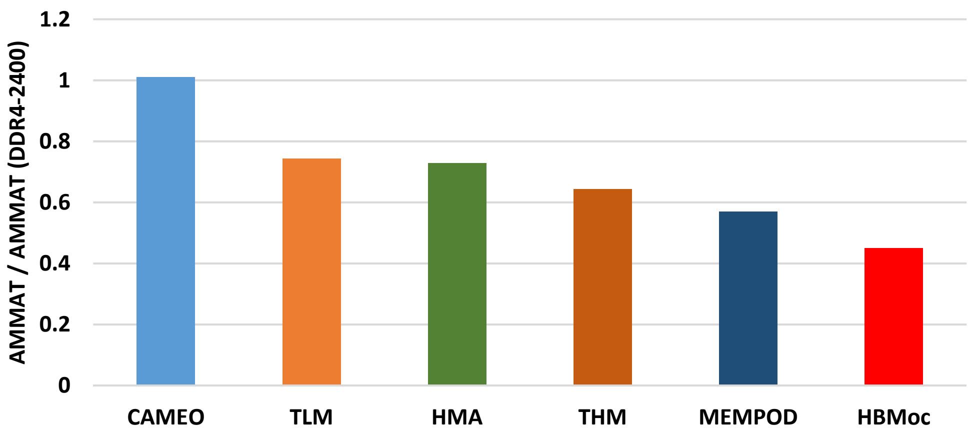


Figure 2.10. Scalability to faster memories. AMMAT normalized to a DDR4-only memory.

We also expect the latency differential between main memory levels to continue to grow. This will happen as 3D memory parts mature, and as we integrate new memory technologies into the system (e.g. hybrid volatile and non-volatile memory systems). To examine this, we model a system where both the 3D DRAM and the DDR memory are faster, but the 3D DRAM is accelerated further resulting in a higher latency ratio between the two. In particular, we simulate a 4GHz HBM as our stacked memory and a DDR4-2400 as our off-chip memory. Since we are modelling a future system, we reduced the fixed penalty for HMA's sorting process from 7ms to 4.2ms (40% reduction) in order to model future faster processors. We assume no caching effects in this experiment.

Figure 2.10 shows our AMMAT results, normalized to a configuration with 9GBs of off-chip DDR4-2400. The label "HBMoc" shows a 9GB configuration with overclocked HBM only. We first observe that CAMEO now reports a 1% AMMAT degradation. The increase in speed differential between stacked and off-chip memory appears to be beneficial for CAMEO, however we can still observe the impacts of intra-segment conflicts. Compared to a configuration with no migration support (TLM), HMA improves AMMAT by 2%, THM by 13% and MemPod by 24%. The overclocked HBM-only configuration is 40% faster compared to TLM.

2.7 Conclusions

MemPod is a scalable, modular and efficient dynamic memory management mechanism. Our analysis demonstrates significant gains compared to state-of-the-art proposals. The modular design achieved with the use of Pods allows for a more scalable migration mechanism while at the same time enforcing small limitations on migration opportunities.

MemPod uses MEA counters to track page access activity and identify hot pages. They are dramatically smaller than prior tracking mechanisms while capturing activity counts and temporal recency in a way that provides more effective prediction of future page access.

Acknowledgements

Chapter 2, in full, is a reprint of the material as it appears in the proceedings of HPCA 2017. Prodromou, Andreas; Meswani, Mitesh; Jayasena, Nuwan; Loh, Gabriel; Tullsen, Dean M., MemPod: A Clustered Architecture for Efficient and Scalable Migration in Flat Address Space Multi-level Memories, 23^{rd} IEEE Symposium on High Performance Computer Architecture (HPCA), February, 2017. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Platform-Agnostic Learning-Based Scheduling: Deciphering Predictive Schedulers for Heterogeneous-ISA Multicore Architectures

In this chapter, we apply a variety of machine learning techniques to the problem of cross-core performance prediction, where the target core potentially differs in terms of its CPU microarchitectural traits, cache organizations, ISA, inorder/out-of-order execution semantics, vector support, and floating point support. The inputs to our predictors consist of application-specific features (obtained via profiling on a "reference" core), and target core-specific features. The output of the model is a performance prediction for the application and target core pair.

Our goal is to not just make effective predictions, but to gain particular insights about the salient features of predictive ML models, the feature correlations (program characteristics, microarchitectural measurements) that are picked up and amplified by the best ML models, and those that are less important. We do this by deciphering (reverse-engineering) our most accurate predictive models. This approach allows us to probe a fundamental question – to what extent can machine learning models understand, analyze, and project the execution of code on arbitrary processor hardware?

Due to the vast number of possible CMP combinations in our database, we present a meticulously developed methodology utilizing statistical analyses and a variety of evaluation frameworks that allow us to derive insights and statistical expectations regarding prediction accuracy and scheduler efficiency.

3.1 Background

Heterogeneous multicores employ cores that can vary in microarchitecture and even instruction set architectures, on the same processor chip. These architectures allow a running application to dynamically migrate execution to the most suitable core, reacting to phase changes and changes to the dynamic execution environment, and further maximizing performance and energy efficiency. While single-ISA heterogeneous multicores [62, 63, 37, 3, 4, 42, 1, 2] offer cores that vary in their microarchitectural configurations to cater to the diverse execution characteristics (e.g., instruction-level parallelism, cache access patterns, branch behavior, etc.) of mixed workloads, heterogeneous-ISA architectures [32, 108, 106, 10, 13, 81, 6, 5, 64, 105], capture an additional dimension of heterogeneity – ISA affinity – the inherent preference of an application code region to execute on a particular ISA. Multiple studies have shown that exploiting ISA affinity could result in significant ($> 30\%$) additional gains in performance and energy efficiency.

However, as the amount of on-chip heterogeneity increases, intelligent scheduling becomes more crucial, since the potential performance and power benefits arrive from smart job allocation. In fact, subpar job allocation can waste resources and power.

State-of-the-art schedulers employ either analytical [31, 25, 78] or statistical [24, 115, 9] (for example, machine learning) predictive models that strive to make accurate performance predictions, and thereby enable near-optimal scheduling decisions. In both cases, predictions are derived using a mathematical equation (or model) whose inputs are typically associated with weights that signify their importance. Analytical models typically employ linear equations, similar to those used by linear regression models. However, these models rely heavily on domain expertise to identify correlations between input features (e.g., instruction mix) and the prediction target (e.g., performance), and consequently determine the appropriate set of feature weights. These models are typically confined to use only a handful of features due to the exponentially increasing complexity of identifying such correlations. Statistical models on the other hand, are trained in software using sophisticated machine learning algorithms that can not only handle far more input features, but can further adapt to non-linear correlations.

Recent technological advances have increased the availability and popularity of machine learning algorithms and tools, leading to an increased number of ML-based computer architecture proposals [24, 115, 9, 113] in recent years. Researchers have demonstrated that predictive models can be deployed and used in modern computation environments, from heterogeneous CMPs to datacenters [25, 78]. However, scheduling for *general-purpose* heterogeneous-ISA CMPs has not been extensively studied, despite significant opportunity for greater performance and energy efficiency. This is in part because, as core diversity increases, the problem of cross-core performance prediction and subsequently the problem of optimal job allocation grows more complex. In a homogeneous multicore architecture, a running thread is expected to have similar performance when it executes on a different core. However, in a single-ISA heterogeneous multicore, the performance difference of the same code region can be vastly different on different cores depending upon the microarchitectural diversity. ISA heterogeneity adds an extra degree of freedom, resulting in a more complex set of parameters that could now potentially affect performance.

3.2 Related Work

This work lies in the intersection of machine learning and computer architecture: We seek to quantify the influence of predictive models and their accuracy, on the final product; a scheduler for heterogeneous-ISA architecture.

We explore three ML algorithms: (a) ridge regression [44] (RR) which is a variant of linear regression [79], (b) decision trees [76] (DT) that are capable of exploiting non-linear relationships by creating a binary decision tree with one condition on a single feature in each node, (c) random forests [17] (RF) that create a collection of decision trees during training and when queried for a prediction, return the average response from all trees. To identify the correlations picked up by our best models, we use two commonly used exploration techniques: linear coefficient analysis, and feature impurity analysis [75]. We further develop a new analysis for categorical microarchitectural features, named "Reverse Path Purity". We describe these techniques in more detail later in the paper.

Single-ISA heterogeneous architectures have been proposed by Kumar et al. [62]. Besides a large body of research proposals, we already see processor manufacturers offering heterogeneous products [37, 53, 104, 26]. There is a large body of research on scheduling and resource management on single-ISA heterogeneous CMPs (such as [101, 31, 98] and more). General-purpose Heterogeneous-ISA multicore architectures [32, 108] include microarchitecturally heterogeneous cores that implement different ISAs. Barabalace, et al. [11, 10] propose Popcorn Linux, an operating system that allows running shared memory applications on a general-purpose heterogeneous-ISA system and further assists in execution migration between the cores. Our work assumes similar software support. Furthermore, exploiting heterogeneous-ISA architectures has been shown to improve security against code-based attacks, such as Return-Oriented Programming, by frequently switching ISAs [107].

Systems combining CPUs and GPUs (or accelerators) in the same package are also examples of heterogeneous-ISA architectures. Our work focuses on systems with general-purpose CPU cores implementing ARM's Thumb ISA[66], Alpha [30] and Intel's x86-64 [45]. Accelerator-based systems (including GPUs) are out of the scope of this work, since the interaction between software and hardware varies greatly between CPUs and GPUs. However, various prediction-based mechanisms have been proposed for such systems [74], often utilizing ML algorithms: Ardalani et al, propose XAPP [8], an ensemble prediction mechanism, which given CPU source code can predict whether this code would benefit from porting to a particular GPU architecture. Hayashi et al. propose a mechanism based on Support Vector Machines

(SVM), which predicts if a given code block should run on a CPU or a GPU for better performance [39].

LACross [115] is a cross-platform scheduler that uses the LASSO linear regression algorithm to predict a phase's performance and power consumption. LACross is evaluated on two heterogeneous machines and enables efficient resource management. Chen et al. [24] utilize Principal Component Analysis for dimensionality reduction and clustering of similar applications for scheduling purposes.

Scheduling on general-purpose heterogeneous-ISA CMP architectures has not been investigated to the same extent as single-ISA or CPU-GPU heterogeneous architectures. Popcorn Linux implements a scheduler that relies on application instrumentation and generates a mapping of application's functions to cores [11].

3.3 Limitations in Prior Work

We identify some common limitations in prediction-based scheduling proposals we aim to address with our methodology: Systems under test are often predefined and do not change throughout each proposal, without discussing if and how the proposed scheduler adapts to other systems. A closely related, frequently-observed drawback is the use of small datasets on which these predictors are trained and evaluated. This is due to the difficulty in obtaining large datasets, particularly if the data is accumulated via simulations (slow). In studies where profiling is used, the hardware cannot be significantly varied, so a large dataset would require tens of thousands of benchmarks.

Several prior studies select one machine learning algorithm, often a linear model, which according to our findings is not necessarily the best option for most cases. Discussion of other available algorithms and how well they perform in the presented use cases is often omitted. Aside from the limited exploration of available algorithms, researchers often pre-select the input features of their models. This selection tends to reflect the collective knowledge of this field and includes variables that have been identified in the past to track dynamic characteristics. However, with modern execution environments we are able to collect significantly more measurements, especially if simulations are used. Furthermore, even though feature selection often requires significant skill and time investment, some ML algorithms are particularly good at filtering features and internally discarding those that have no impact on the final prediction. Finally, ML-based studies that show high prediction accuracy do not explore the machine's "reasoning" when making decisions. We seek to understand what these models consider important when predicting an application's performance on a core.

In this work, we address drawbacks commonly found in prior work: To address the issue of small datasets we simulate 72 workloads on 600 different cores each, for a total database size of 43200 entries. The cores we include span 3 different ISAs: Thumb, Alpha, and Intel's x86. The large heterogeneous-ISA core collection is shown to result in predictive models that generalize and adapt to a variety of underlying systems without modification. We explore thousands of configurations of each ML algorithm (different optimizations, input sets etc) and identify the optimal models. For the majority of this work, we do not attempt to pre-select the desired features, instead we allow the ML models to use them all as they see fit. Once our models are trained and report acceptable prediction accuracy, we seek to understand how these accurate predictions are achieved. We use an in-house scheduler evaluation framework to describe the relation between ML metrics (such as mean error, standard deviation etc) and the efficiency of ML-based schedulers. We demonstrate that when we use ML models as the predictive unit of a scheduler, isolated ML metrics do not necessarily reflect the scheduler's quality.

3.4 Prediction Accuracy Vs Scheduler Quality: An In-Depth Evaluation

Schedulers clearly benefit from accurate performance predictions. Depending on the system under test, trial and error scheduling on a complex system (many threads, diverse cores) is (1) unlikely to find the optimal schedule and (2) likely to sample many poor schedules before finding good ones. A scheduler that can predict the performance of any given application on any core can intelligently distribute jobs for maximal throughput without the overhead of sampling all possible permutations.

Intuitively, we expect that a system's level of heterogeneity can affect the difficulty of the scheduling problem. In other words, it should be easier to create good schedulers for systems with lower diversity. In case of predictive schedulers (ML-based or otherwise), prediction error should have a smaller impact on realizing efficient schedules on less diverse systems. Although intuitive, to the best of our knowledge, the impact of heterogeneity with respect to the scheduling problem has not been quantified in the literature. Our **Ease-of-Scheduling** (EoS) metric has been developed in response to this ambiguity.

Regardless of the underlying system, all prediction units can be characterized by their prediction accuracy, using the mean and standard deviation of their prediction error. However, depending on the amount of underlying heterogeneity, these metrics do not necessarily provide an accurate image as to the expected performance of the derived scheduler that uses this unit. Our **Expected Scheduler Efficiency** (ESE) metric statistically quantifies how accurate a performance predictor needs to be in order for the scheduler to lead to high performance, given a system with some arbitrary scheduling difficulty (a system with a given EoS).

Finally, we use αIPC to maintain fair performance comparison between workloads compiled for different ISAs. Since the number of dynamic instructions varies between ISAs, we measure execution progress in *Alpha* Instructions Per Cycle. Our compilation framework allows us to map regions of executables to their corresponding regions in an Alpha executable, thereby enabling this metric.

3.4.1 Ease of Scheduling (EoS)

We measure EoS by evaluating a system that uses a *random* scheduler. Workloads are run to completion and when execution completes, we compare the obtained mean αIPC against

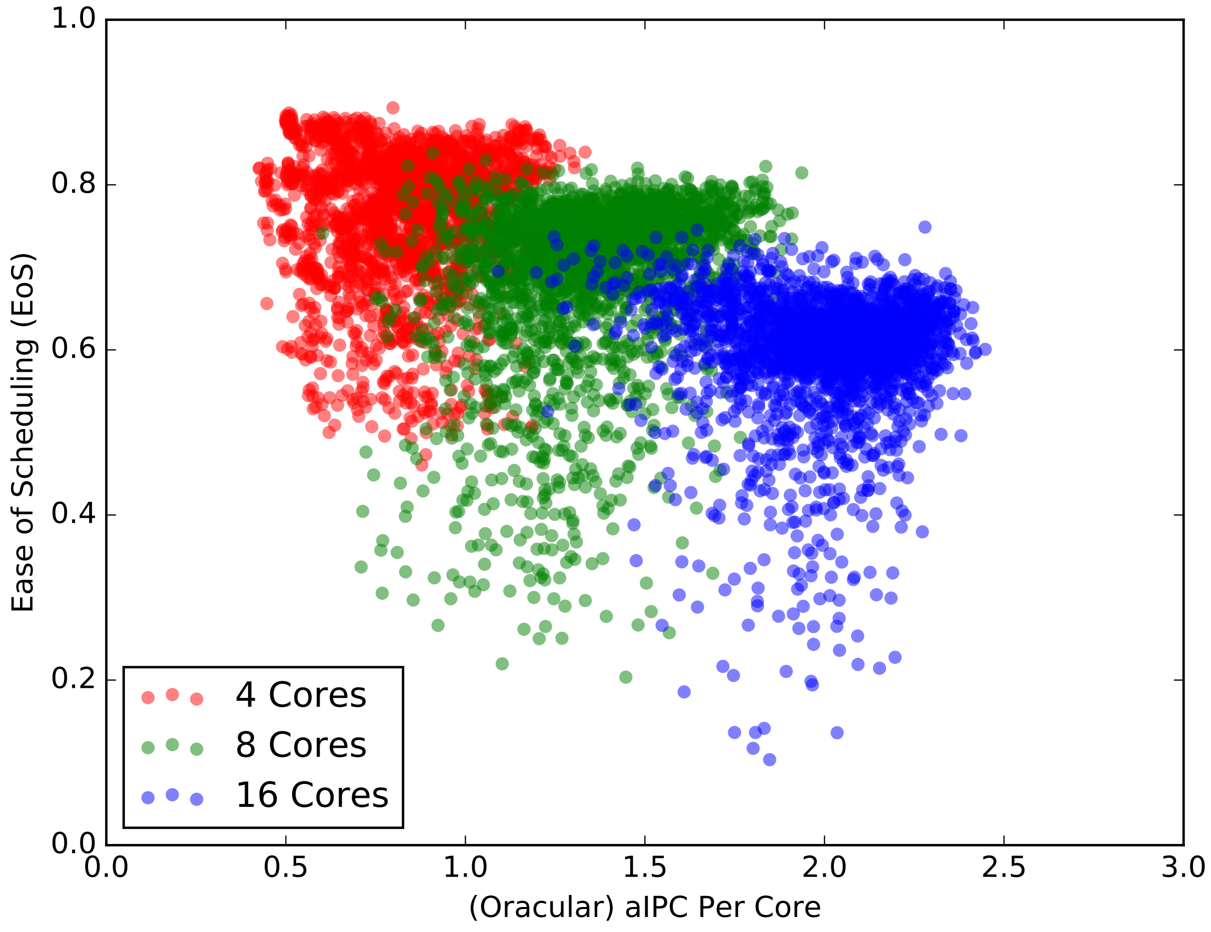


Figure 3.1. Ease of Scheduling analysis

the mean αIPC the same system reports using a "monte-carlo optimal scheduler". The closer the random scheduler scores compared to optimal, the easier it is to schedule for the system under test. Higher scores mean easier systems.

To minimize noise caused by random decisions, we execute each experiment 300 times, with 200 workloads scheduled per execution and report mean performance. The workloads are randomly chosen *before the experiment begins and remain constant throughout the entire experiment,* for all 300 runs on each of the 7500 systems under test. We ignore cross-core migration overhead since our predictive models don't yet consider cross-ISA binary translation and state transformation costs, which can significantly vary with different application characteristics and different target ISAs. More details on our experimental methodology are presented later in Section 3.5.

With 200 workloads, the problem of finding a globally optimal schedule is practically infeasible. The number of possible schedules for a 4-core system is 270 digits long. We define optimal performance for each system, as the maximum *observed* performance from all our EoS and ESE experiments. Combined, we compare almost 14 thousand scheduling options on all systems, while scheduling the same 200 workloads appearing in the same order. Throughout the experiment, we use a random scheduler, as well as schedulers based on predictors of varied statistical accuracy, in terms of their Mean Absolute Error and Standard Deviation.

Figure 3.1 presents our EoS analysis on 7500 randomly chosen heterogeneous-ISA multicore systems. More accurately, we present results on 2500 randomly chosen 4-, 8-, and 16-core systems. Our results highlight the impact of various factors on a system's EoS. First, we observe that the number of cores appears to have an impact, regardless of the system's level of heterogeneity, with each cluster hitting a lower EoS ceiling than the previous one. We also observe significant intra-cluster diversity for all clusters. Our results show that the most difficult systems to schedule resemble a Cell-like heterogeneous-ISA architecture. For example, the most difficult (lowest EoS) 8-core system, contains one high-end x86 core, two medium cores (1 Alpha, 1 x86) and five low-performance, microarchitecturally heterogeneous Thumb

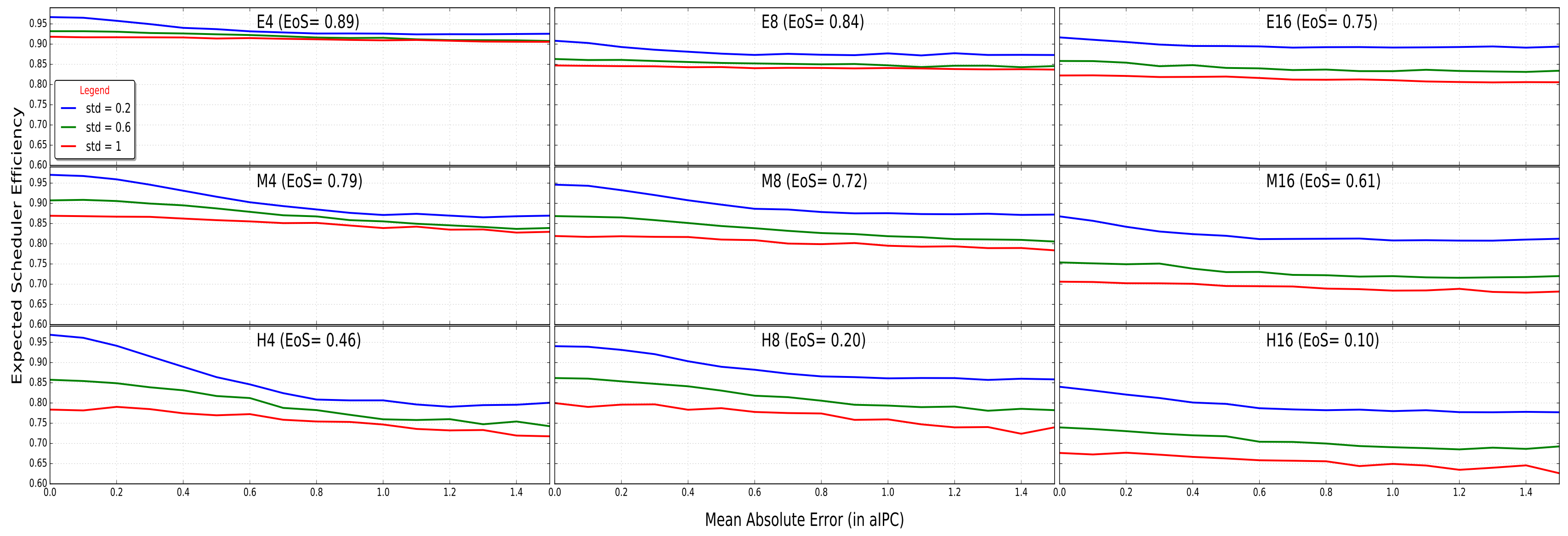


Figure 3.2. Scheduler efficiency on Easy (E), Medium (M) and Hard (H) 4-, 8-, and 16-core systems.

cores. Such systems appear to be very unforgiving towards prediction error since there are statistically, limited options that provide optimal or near-optimal performance, each time the scheduler is invoked to make a decision. The easiest systems tend to have more balanced (in terms of performance) cores. Our easiest 16 core system for example, has three high-end x86 cores, while all others are medium performance Alpha, Thumb, or x86.

We use EoS to select benchmark systems for the remainder of this paper. Due to space restrictions, we cannot present all architectural details for each core in the system, unless we limit our study to a very small number of systems. In this work, we consider benchmark systems, characterized by their EoS instead of their architectural traits. Specifically, we use three 4-core, three 8-core, and three 16-core benchmark systems. For each category we use the easiest, medium, and most difficult system. With our benchmark systems ranging from maximum to minimum EoS, and from 4 to 16 cores, we cover a wide range of heterogeneous-ISA multicore systems.

3.4.2 Expected Scheduler Efficiency (ESE)

ESE is defined as the ratio of average system αIPC over the average αIPC of an optimal scheduler, given a predictor characterized by some mean prediction error and standard deviation (of prediction error) values. We find that more accurate predictions do not necessarily translate into more efficient schedulers. Instead, the system's EoS can have a significant impact, as well as the predictor's accuracy uniformity (standard deviation of prediction error).

In this experiment, we evaluate a typical predictor-based scheduler on each of our 9 benchmark systems. We perform a sweep over possible predictive units by varying Mean Absolute Error (MAE) values from 0 (very accurate) to 1.5 (high prediction error) αIPC and we use three different standard deviation (STD) values (low, medium, high). Prediction error is randomly drawn from a normal distribution characterized by MAE and STD, and applied on the real (oracularly obtained from cycle-accurate simulations) αIPC values. The assumed scheduler receives predictions and decides on a *greedily optimal schedule* that maximizes overall αIPC .

We negate absolute error values (since we use MAE), based on the outcome of a fair coin flip before we add it to true performance. Once again, we perform 300 runs with 200 workloads scheduled in each run to measure average αIPC for each data point.

Figure 3.2 presents our ESE evaluation results. We label each of our benchmark systems using 'E' (for Easy), 'M' (Medium), and 'H' (Hard), followed by the number of cores. For example, “M16” is a 16-core system of medium scheduling difficulty as identified by our EoS analysis. We also include the EoS value of each system in our results. We first observe that standard deviation demonstrates equal or even higher impact on ESE than MAE. Increased system difficulty further exacerbates this effect. For example, the H4 system's ESE drops by up to 17% within the MAE range, but by 19% within the STD range. We further observe that even in the presence of predictors with very high accuracy and uniformity, the system itself dictates how efficient the final scheduler can be, with statistical drops over optimal scheduling of 8%, 13%, and 16% for our 16-core (E,M,H respectively) benchmark systems. On the other hand, the easier a system is, the more immune it becomes to prediction error, with ESE lines not varying much as MAE increases.

Lastly, we find that all our systems demonstrate a significant error tolerance slack: We observe no more than a 2% ESE drop for MAE values under 0.3 and STD=0.2 across all systems. Furthermore, as STD increases, MAE slack also increases, since the predictor is closer to random and eventually plateaus to a low expected efficiency. Beyond the system's EoS level, it is worth noting that even with 100% accurate predictions, a greedy scheduler (locally optimal decisions, no knowledge of the future) cannot guarantee optimal scheduling.

In conclusion, based on our results, and assuming full and extended system utilization, we argue that ESE analyses can unveil tradeoffs during the design phase of future heterogeneous-ISA CMP systems. It is possible that system peak performance can be traded off in order to ensure more uniformity during execution, as well as cost savings on scheduler development. Both EoS and ESE metrics are computationally easy and cost effective exploratory tools.

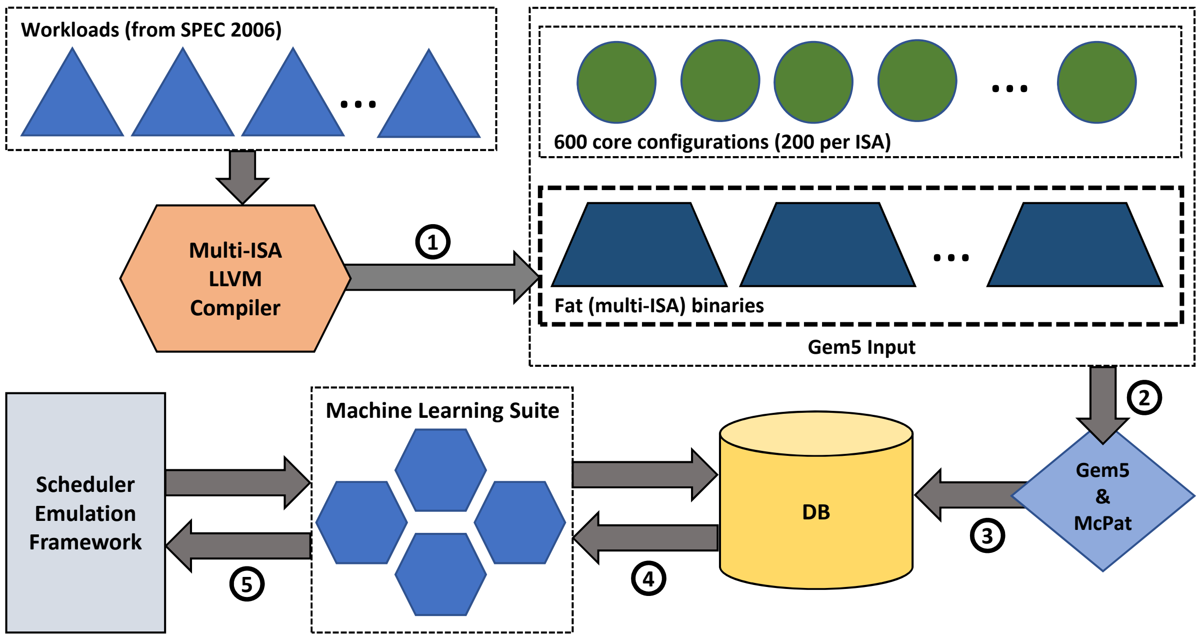


Figure 3.3. Experimental Framework overview

3.5 Experimental Framework

This study requires a diverse set of evaluation frameworks. First, to train and evaluate machine learning models, we develop an in-house ML suite which allows us to explore a variety of ML models in depth. Second, to assess the capabilities of trained ML models when used within schedulers, we implement a runtime scheduler framework that makes greedily optimal decisions periodically.

We use the simulation-based dataset collected during the design-space exploration published by Venkat et al. [108]. Figure 3.3 presents a high-level overview of our experimental framework, including the toolchain flow for creating the dataset. Our frameworks allow us to efficiently explore the interaction between ML models and multi-ISA software and hardware.

3.5.1 Machine Learning Suite

The Machine Learning aspect of this work is a multi-dimensional problem. On one hand, we seek to explore the behavior of various ML algorithms, with each having its own set

Table 3.1. Configuration flags description (ML Suite)

Flag	Description
Scale	Performs data scaling
Whiten	Performs data whitening
RefCore	Reference core selection (1-600)
KeepRAW	Keeps features with raw values (e.g. INT_R)
KeepPower	Keeps profiled dynamic power features
KeepL1	Keeps L1 cache features
KeepL2	Keeps L2 cache features
Target	Sets the prediction target (typically, aIPC)
KeepRefTarget	Keeps the profiled target (aIPC) value
KeepFI	Keeps Fetch/Issue rate measurements
TreeDepth	Used with tree-based algorithms

of parameters that could potentially improve its accuracy if set at proper values. On the other hand, we work with a significantly large set of features, on which we need to have fine grain control across experiments. For example, we want to have the ability to study how each ML algorithm reacts if we stop providing cache-related features (capacity, number of hits and misses, etc), branch misprediction statistics, etc. Our ML suite supports all our requirements and ensures a fair comparison across all the ML models explored in this work. At its core, our ML suite utilizes the sklearn Python library [93].

The ML suite supports a variety of boolean flags for enabling and disabling feature groups (e.g. cache information) at the model's input, as well as ML optimizations (data scaling and whitening). Table 3.1 presents an overview of all available flags. Besides control over the input and optimizations, we also have the ability to select which ML algorithm will be used and what kind of training will be performed (i.e. how our database will be split into train and test sets - discussed in Section 3.7).

Once the parameters are populated with the desired values, our suite includes functions to train, test and score each model. Due to the multiple control options in our study, each machine learning algorithm we explore can have 1024 different configurations (10 boolean flags). Tree-based algorithms have 5120 flavors because of the tree depth variable.

3.5.2 Scheduler Framework

We developed an event-driven scheduler framework, in order to compare predictors, when used within schedulers.When studying ML-based schedulers, our framework is directly linked to the ML suite presented earlier. The model-under-test is first restored and trained. Once it is ready, our scheduler queries the ML suite to retrieve predictions when necessary. Non-ML-based schedulers such as oracular and random, are implemented directly in our framework.

To compare predictors, we first configure the target system by selecting cores from our database using their unique identifier. We then select and if necessary, train the predictor, before scheduling begins. We designed our scheduler to mimic an execution scenario where workloads appear one at a time as soon as an in-flight workload finishes. In other words, at any given time, the number of in-flight workloads equals the number of cores in our system. Our scheduler's goal is to greedily (no knowledge of future workloads) schedule these workloads in order to achieve the maximum sum of αIPC across all cores, according to the αIPC predictions it receives. Once a schedule has been identified, workloads are migrated to their new cores and resume execution.

In this work, we assume that all our workloads have been profiled on one reference core, which is not necessarily part of the underlying system. Our ML models are trained to provide "one-to-all" predictions. In other words they extrapolate application behavior on the reference core, to the target core. During evaluation we provide our models with a choice of two medium-performance reference cores, one Alpha and one x86, and we allow them to select one to be used in future experiments. Our methodology can be adapted to use one or more of the system's cores as reference, as well as train more specialized predictors for each pair of underlying cores. We choose to use one predefined reference core for two reasons: (1) we can better study the generality of our models in a one-to-all prediction setting, and (2) training more specialized, pairwise predictors would require severely reducing our dataset (to only include two cores instead of 600).

Table 3.2. Breakdown of workloads in our database.

Benchmark	# of Phases
bzip2	9
gcc	8
gobmk	8
hmmer	5
lbm	1
libquantum	7
mcf	9
milc	9
sjeng	7
sphinx	9

3.6 Database Description and Data Optimizations

This section describes the database we use throughout this work and our data splitting methodology. Table 3.2 lists all the workloads (benchmark phases – 100M instructions each) in our database. Table 3.3 presents the configurations of the cores in our dataset. A justification is presented on why these particular configurations are chosen out of a vast set of possible – but often unrealistic – microarchitectural options, in the original proposal [108].

Each database entry consists of 48 features. Of those, 30 are workload characteristics obtained via gem5-based profiling. The remaining 18 characterize cores and are statically available features. We refer to these two sections of our database as the "workload" and "core" sections.

3.6.1 Workload and Core Sections:

Dynamic workload features can vary according to the core they execute on; for example, instruction count will be constant within an ISA, but different across ISAs, while cache misses will vary by core features and ISA. Table 3.4 presents a breakdown of these features, grouped for brevity.

We characterize each core using 18 features. Each feature is collected from the core's microarchitectural specifications and is available from the manufacturer. Table 3.5 presents all

Table 3.3. Description of core configuration points.

Design Parameter	Design choices
ISA	Thumb, Alpha, x86-64
Execution Semantics	In-order, Out-of-Order
Branch Predictor	Local, Tournament
Reorder Buffer - Register File **(ROB - Int. regs - FP regs)**	64-96-64, 128-160-96
Issue Width - Functional Units **(Width - Int. ALUs - Int. mult - FP ALUs - FP mult - SIMD)**	1-1-1-1-1-1 1-3-2-2-2-2 2-3-2-2-2-2 4-3-2-2-2-2 4-6-2-4-2-4
Load Store Queue Sizes	16, 32 entries
Cache Hierarchy **(L1 - L2 - L3)** **32K/4 → 32KB, 4-way**	32K/4 - 32K/4 - 4M/4 32K/4 - 32K/4 - 8M/8 64K/4 - 64K/4 - 4M/4 64K/4 - 64K/4 - 8M/8

the core features in our database. Some of our features are categorical, without an intrinsic order, for example a core's ISA. In other words, they cannot be ordered from best to worse, smallest to largest etc. Such features are a problem when used with certain ML models. "1-hot encoding" is a commonly used technique that provides a simple solution to this problem. Ordinal features such as cache sizes do not present this problem.

3.6.2 Data Splitting

Our decision to use benchmark phases as workloads complicates how we split our database into train and test sets for correct evaluation. If we follow the commonly used ratio split method (database entries randomly assigned to either train or test set), we introduce information leakage between the two sets: if some workload A, runs on CPUs 1 and 2, these two database entries should not be split across train and test sets, because the trained predictor would have full information about workload A when making predictions. Similarly, workload A and workload B, on any pair of CPUs, where A and B are different phases of the same benchmark, should also not be split across train and test, because the phases may actually share some code, and certainly share the dataset.

We use our ML suite to study both these problems: When using random data split, we observe excellent prediction accuracy (0.006 MAE) over the (supposedly) unseen test set, which is of course a consequence of this particular splitting approach and not a representative result. When we assign all 9 phases of the *mcf* benchmark as our test set (all other benchmarks in train set), we measure 0.48 MAE, which is quite high. Moving just one mcf phase from test to train, MAE immediately drops to 0.16.

A better way to split data in such cases is to use a cross validation technique named Leave-One-Group-Out (LOGO) [92]. Following the LOGO strategy, one group is assigned as the test set, while all other groups form the train set. For correct evaluation under this technique, training and testing are performed in a loop, with each group assigned as the test set at least once.

To solve the information leakage problem caused by our workload definition and increase the generality of our evaluation, we define LOGO groups as entire benchmarks, with all their phases, including runs on all core types. Training and testing is performed over all groups for each ML model and we measure MAE across all iterations. We use single-benchmark LOGO for this exploratory phase of our work.

3.7 Deciphering ML Models

3.7.1 ML Model comparison

We perform a brute-force exploration over the space generated by our configuration flags (Table 3.1), to identify the best variation for each of our three algorithms.

We rank the top models based on their prediction accuracy (Mean Absolute Error) over the test set. To study the impact of the reference core's ISA, we use a mid-range Alpha as well as its equivalent x86 core as reference cores (an equivalent core is one where the microarchitecture features are constant, only ISA-specific differences). Our test sets still contain cores from all 3 ISAs, so regardless of the reference core chosen, all our models make αIPC predictions for all other cores in our database. Our ML framework allows us to pick any of the 600 cores as the reference core.

Table 3.4. Dynamically-collected feature groups in our database.

Feature Names (grouped)	Description
APPID	Unique workload identifier. Never used for testing or training
INT_R, FP_SIMD_R, BR_R, LOAD_R, STORE_R	Dynamic count of five instruction types: Int, fp or simd, branch, load, store
INT_N, FP_SIMD_N, BR_N, LOAD_N, STORE_N	Normalized dynamic instruction count
OPS	Total number of operations. Varies across ISAs
<TYPE>_HITS_R, <TYPE>_MISS_R, <TYPE>_MISS_N	Raw count of hits and misses, and miss ratio. TYPE=$I (I-cache), $D (D-cache), L1 ($I & $D), L2. Total of 12 features in this group.
FETCH, ISSUE	Dynamic fetch and issue rates
REF_IPC	Performance measurement (in aIPC)
MISSPRED	Branch missprediction rate
D_PROC_PWR, D_CORE_PWR	Dynamic processor and core power consumption (McPat)

Table 3.5. Statically-collected core features in our database.

Feature Names	Description
CPUID	Unique core identifier. Never used for testing or training.
ISA (THUMB, ALPHA, X86)	Core's ISA. Categorical feature.
SEMANTICS (IO, OOO)	In-order or Out-of-Order. Categorical feature.
BR_PRED (LOCAL, TMNT)	Core's branch predictor, local or tournament. Categorical feature.
WIDTH	Core's width
ROBSIZE	Reorder buffer size
INTREGS, FPREGS	Number of int and fp registers
LSQSIZE	Load-Store Queue size
L1SIZE, L2SIZE	Cache sizes for L1 (kB) and L2 (LLC, MB)
INTALUS, FPALUS	Number of int and fp ALUs
INTMULTDIV, FPMULTDIV	Number of int and fp multiply-divide units
SIMD	Number of SIMD units
AREA	Core's area (McPat)
PEAKPOWER	Core's peak power (McPat)

Table 3.6 presents our evaluation results as well as the configuration of each model. we present MAE, and STD results of each model when queried on their train set entries (all previously seen data), in addition to showing results when queried with the test set. Without loss of generality, we can expect any scheduler to occasionally be confronted with the same applications, possibly even similar inputs – querying with train set entries gives an upper bound to the quality of the predictions when a new workload closely matches a previously seen one.

On the configuration side, the top models have some interesting disagreements. First, an Alpha reference core is preferred by the RR model, while the tree-based models prefer the x86 core. For comparison, the equivalent RR model that uses the x86 reference core, reports 5.4x higher prediction error over the test set and the most accurate RR model that uses the x86 reference core shows 13% lower prediction accuracy compared to the top RR model. Second, our three models also disagree on whether to keep dynamic power consumption measurements, with only the RF model utilizing this information. Third, the RR model scales its input features and drops all raw measurements (very high values) in an attempt to keep all its feature values roughly in the same range which is a common optimization for linear models. Finally, the only features kept by the RR model in addition to normalized dynamic instruction breakdown (cannot be turned off by any model) are L1 cache and reference target features (profiled αIPC on reference core). We later explore which of these features our models consider important and how much weight they are assigned on the final prediction.

Interestingly, the DT model chooses to ignore L1 cache features and the αIPC measurement from the reference core and instead works with raw values, measured fetch and issue rates, and L2 cache features. No optimizations are applied since the DT algorithm shines at internally dealing with messy data. The RF model keeps the highest number of features, but ignores all cache information. It also chooses to whiten its features. This is because keeping correlated features reduces the variance between the forest's trees and as a result reduces prediction accuracy. For comparison, the equivalent RF model without whitening reports a 6% higher MAE.

All three of our models enjoy a very large training dataset, due to the LOGO splitting

Table 3.6. Comparison of top ML Models

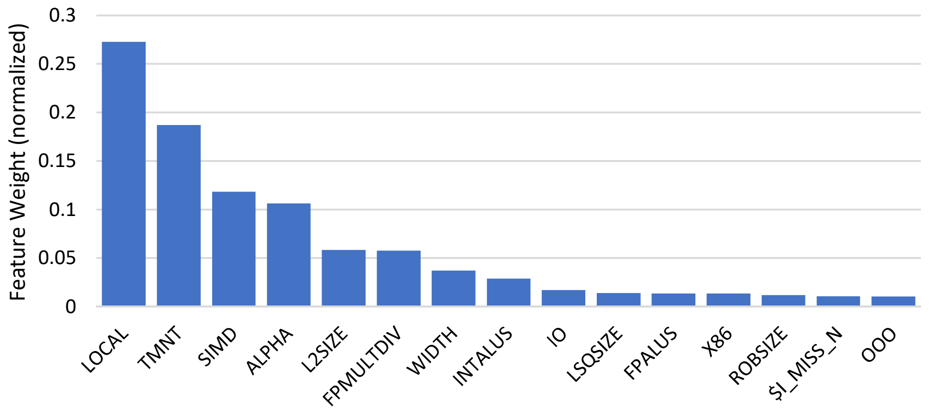
Model	Test Set		Train Set		Configuration of best model after full space exploration										
	MAE	STD	MAE	STD	Scale	Whiten	Ref Core	Keep RAW	Keep Power	Keep L1	Keep L2	Keep Ref. Target	Keep FI	Tree Depth	#Trees
RR	**0.23**	**0.3**	**0.21**	**0.26**	✓	✗	Alpha	✗	✗	✓	✗	✓	✗	--	--
DT	**0.22**	**0.35**	**0.07**	**0.04**	✗	✗	x86	✓	✗	✗	✓	✗	✓	12	--
RF	**0.19**	**0.31**	**0.03**	**0.04**	✗	✓	x86	✓	✓	✗	✗	✓	✓	11	10

technique. Such a dataset is not unreasonable in a production-level scheduler, given periodical re-training with newly observed workloads. However, we later (Section 3.9) show that the RR model is much more sensitive to reduction of the train set size, whereas tree-based models remain within error tolerance.

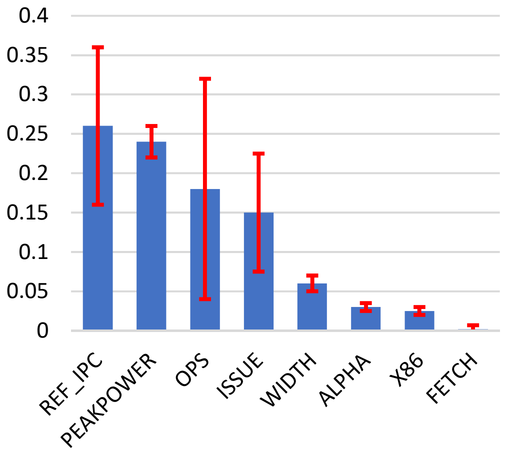
3.7.2 Linear Coefficients (LC) Analysis

The linear coefficients analysis can be applied to linear models and gives us an understanding of how it internally utilizes its input variables. During training, Ridge (linear) Regression algorithms generate an equation of the form: $\overline{\alpha IPC} = \sum_{i=1}^{n} c_i f_i$ where n is the number of features, and c_i is the coefficient associated with the feature f_i. Features with higher *absolute* coefficient values have more impact on the final prediction than those with lower coefficient values. The linear coefficient analysis simply sorts each feature based on its significance. However, features with extremely high measurement values compared to the rest of the input can still have significant impact even with lower coefficients. Since our top RR model applies the scaling optimization, we know that all our features have the same mean value of zero, and standard deviation of one. This allows us to simply sort the coefficients themselves in order to understand each feature's significance.

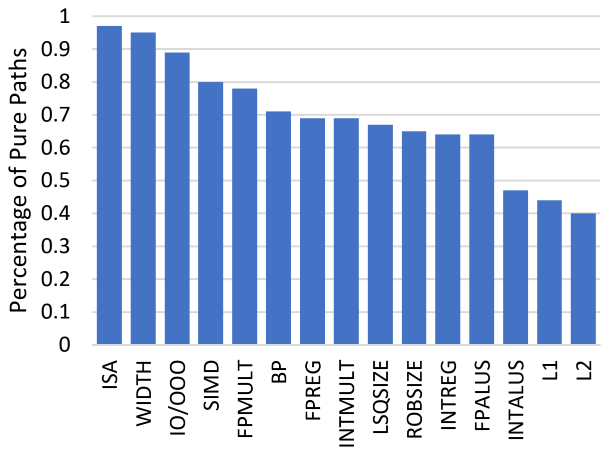
Our results presented in Figure 3.4a reveal that this linear model assigns extremely high weight on the hardware characteristics of the target core. We only present features that have normalized weight of at least 1%. Interestingly, this model assigns significant weight on the type of branch predictor of the target core (LOCAL or TMNT). It also assigns some weight to the target's ISA, as well as its number of functional units, and execution semantics (IO or OOO). When we look at the collection of hardware features that are considered important, it appears that this model is attempting to quantify the computational capacity of its target core. Such an approach is reasonable since this model is only allowed to draw a straight αIPC prediction line. Its best bet is to predict performance based on the overall computational capacity offered by the core. If instead it tried to predict based on dynamic workload measurements or even a



(a) LC Analysis



(b) MDI Analysis



(c) RPP Analysis

Figure 3.4. Normalized feature importance from three different analysis techniques. Refer to tables 3.4 and 3.5 for a description of our keywords. LC analysis performed on top RR model. MDI performed on top RF model. RPP performed on near-oracular DT predictor.

combination of the two, the relationship is likely to be non-linear. We notice that even though the profiled αIPC from the reference core is part of this model's input (refer to Table 3.6), it is assigned a weight less than 1% – in other words it is an insignificant part of the prediction equation derived by our most accurate RR model. Finally, we observe that this model picks one dynamic (profiled) feature and assigns it a small weight – the number of dynamic *instruction* cache misses, which confirms the fact that instruction cache misses, although infrequent, can have a substantial impact on overall performance.

3.7.3 Mean Decrease of Impurity (MDI) Analysis

Feature impurity analysis is performed on tree-based models. We apply it on our top random forest, since a collection of decision trees gives us a broader view of the feature space. During tree construction, these algorithms create decision nodes with each node being a "less than X" condition on a single feature, where X is typically a real number.

Similarly to linear models gradually adjusting feature weights to minimize prediction error, decision node conditions are chosen such that they minimize the "impurity" of the two subtrees generated by that node. Since we are using *regression* models (not classifiers), the impurity measurement is defined as the variance between all final responses (predictions) of each subtree. During training, we can compute how much each feature decreases the weighted impurity in a tree (e.g. how narrow it makes the subtree's prediction range). Since our RF model performs data whitening, we avoid a shortcoming of this analysis, where the presence of highly correlated features gives the appearance of lower importance (because correlated features can be used interchangeably at decision nodes).

Figure 3.4b presents our impurity analysis results. Similarly to Figure 3.4a, we only plot the features with at least 1% of normalized importance. Since we study a *collection* of predictors, we are also able to plot standard deviation lines showing how each feature is ranked amongst all trees.

Very short deviation lines mean that the given feature is considered roughly equally important by the collection of trees, while longer lines imply that the feature's importance varies more between trees.

The RF model assigns high significance to fewer features than the RR model (8 vs 15). We also observe that this collection of features seems more reasonable for performance predictions. For example, if we focus on their average importance, the profiled αIPC is considered the most influential feature, followed very closely by the target core's static peak power (not to be confused with the workload's profiled dynamic power consumption).

Peak power is a very good proxy for the peak performance of the target core – power consumption is a major factor for any commercial core produced, so manufacturers are typically not willing to increase their product's peak power consumption unless it offers significant gains in performance.

Interestingly, we observe significant difference in the amount of deviation in the top two features. While all trees consider peak power equally important (predict based on target core), the importance of the profiled αIPC varies significantly amongst trees. This partially explains the surprising decision of the linear model presented earlier to not use any of these measurements. Even though this feature combination is meaningful and according to our results successful, their relationship towards the prediction target is non linear (we verify this phenomenon by visualizing all our database entries). Random forests on the other hand are able to exploit this non-linear relationship by having only a small set of trees using profiled αIPC while the remaining trees use other measures to minimize the overall prediction error, and at the same time use the peak power feature in all trees. Clearly, this feature combination is very strong given the excellent prediction accuracy of this model on the train set.

We further observe an architectural feature overlap between the RF and RR models – the target core's instruction width and ISA (all with relatively low importance but also short deviation lines). Notice that when a decision tree path asks the two ISA questions sequentially, it is able to differentiate between all three of our ISAs, unlike the linear model which ends up with less information. Instruction width and ISA are among the top features in our best DT model as

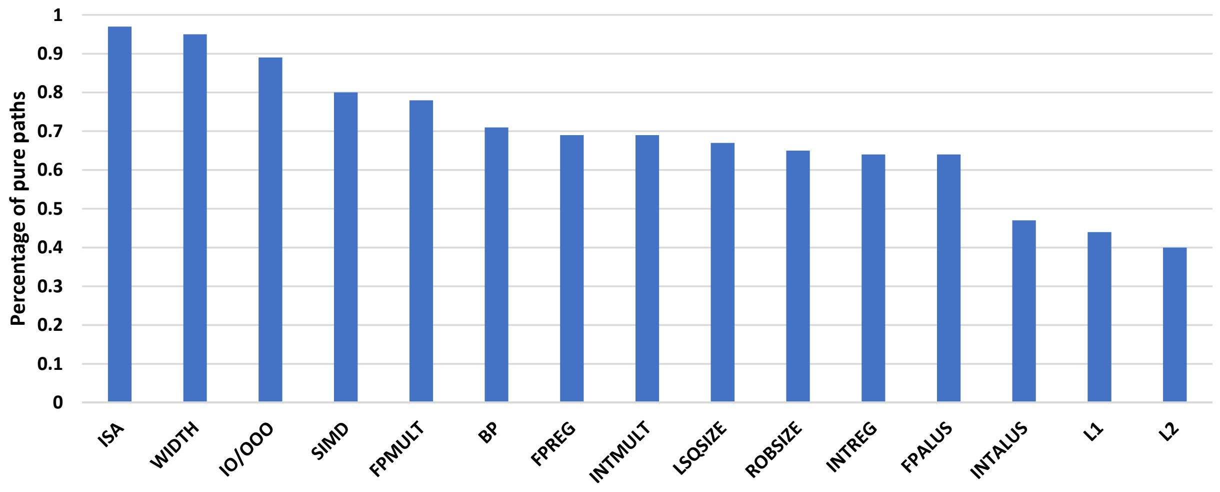


Figure 3.5. Reverse Purity Analysis, applied on 100% accurate DT model.

well (MDI analysis results not shown), showing a strong consensus among all types of predictors on the usefulness of these variables. With the ISA variable playing such an important role, it is clear that prediction techniques focused on single-ISA heterogeneous datasets cannot predict accurately when applications cross ISAs. To the best of our knowledge, we are the first to design predictors that remain accurate across general-purpose ISAs.

We finally observe that this model also focuses on profiled features such as the profiled number of ops (very high deviation), issue, and fetch rates. It stands to reason that fetch and issue rates hint towards the workload's parallelism levels which eventually affect its performance. The amplitude of this effect depends on the target core's characteristics, which are also queried by our RF model.

3.7.4 Reverse Path Purity (RPP) Analysis

Consider a decision tree model which has near-perfect prediction accuracy. This model is essentially a collection of paths that start at the root node and end on leaf (decision) nodes. Each path asks a sequence of questions that "filter" the input query. Intuitively, if we observe that some *categorical* feature X which has N possible values gets cleanly separated amongst the tree's paths, then we can argue that the given model deemed important to always "know" the exact value of X before making a prediction. In this experiment we refer to tree paths as pure, if and only if they can be traversed by exactly one value of some categorical feature. Paths that are shared by two or more values are referred to as impure.

It is challenging to measure path purity by simply traversing all paths and reading their questions, because some features can be extrapolated by questions on other features. For example, our tree can ask “Is the target ISA Thumb?”, or it can ask “Is there floating point support on the target core?”. We devise this experiment to overcome the possibly unpredictable combination of questions that can lead to a single feature value, and it allows us to measure path purity for all features that have a finite number of values.

We begin by creating a near-perfect predictor. We achieve this by training our best DT model using the entirety of our dataset (everything is in the train set). This leads to minimal MAE and STD (both are measured below 0.02). Of course the actual accuracy measurements are irrelevant to this study, but we now know that we have a predictor that successfully converted our entire dataset to 2117 (measured) distinct paths of up to 12 questions each.

In order to measure path purity, we first pick the categorical feature we seek to study. For this walkthrough example, let’s assume we pick the ISA feature. Then, we query this model for a prediction for each row of our dataset. We maintain a map between leaf node ID and a count of different ISA values that reached it; by definition exactly one path can reach one leaf node. Those nodes that only counted a single ISA value are pure, while all others (with 2 or 3 values) are not. Clearly, if the model internally fully separates ISAs to different paths, it is of paramount importance (to the model) to “know” which ISA it’s predicting for, leading us to the conclusion that ISA is an important feature. If on the other hand, some other feature is shared in most paths, then – regardless of the features implicit weight – this accurate model cares less about extracting the exact value, thus the feature is of lower importance.

We plot our findings in Figure 3.4c. We see that a near-perfect model separates the target core’s ISA and width almost entirely (97% and 95% of the paths are pure respectively). Execution semantics (inorder/OoO) follow with 89% path purity. The majority of our core-

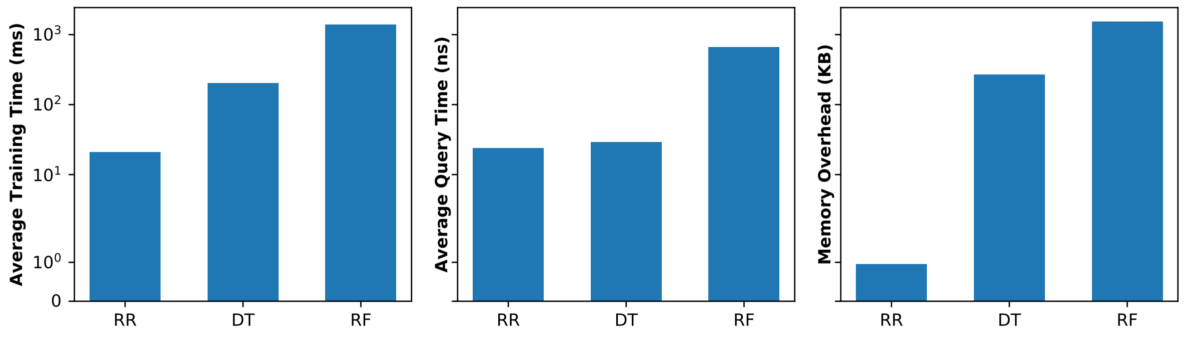


Figure 3.6. Predictor algorithm overhead comparison: Training time (left), Query time (middle), Memory overhead (right).Y axes shared and in log scale.

describing features lies between 80% and 60%, while cache sizes and the number of integer ALUs are below 50% (the majority of their paths are impure). This experiment proves that ISA, width, and execution semantics are of paramount importance when predicting performance for heterogeneous-ISA systems. This finding is in agreement with our insights derived in the previous two experiments, however we now have a clear image of just how important these features are. The reason previous experiments do not show higher weights on some of the features (e.g., the ISA) is because they can often be extrapolated via other features.

3.8 Predictor Overhead Comparison

This section presents an overhead comparison for our three models. We perform the following experiments on a system with an i7-3770K processor running at 2.9GHz and 16 GB of memory. We use the algorithm implementations from sklearn [93].

First, we vary the dataset size and train each model 100 times to report average training overhead. All algorithms are measured to have linear algorithmic complexities for training, however RR has the lowest slope of 1.4, followed by DT (14.3) and RF (92.7). Training times for a dataset of 30k entries are 21ms, 203ms, and 1.4s for RR, DT, and RF respectively (Fig. 3.6 – left). Training overhead has little significance in choosing a model, since it happens infrequently. For extremely large datasets however, RF's overhead might become prohibitive.

We measure query overhead (Fig. 3.6 – middle), by asking each trained model for 55k predictions and report the average time per prediction. Overhead per query for RR, DT, and RF is 24ns, 29ns, and 667ns respectively. Query overhead is an important metric, since predictions can be part of the critical path, especially in the context of job scheduling. RR and DT are comparable, however RF is significantly slower.

Our deciphering methodologies presented in Section 3.7 identify which dynamic measurement each model requires. Dynamic measurements require HW support, specifically on-chip counters. We find that the RR model requires one counter (# instruction cache misses) and the DT model needs 4 counters (# ops, # L2 cache misses, # load instructions and # of branch instructions). Finally, the RF model requires 5 counters (profiled αIPC , issue and fetch rates, # ops and # loads). We note that most modern cores already feature these performance counters.

Finally, we measure the memory overhead of each model (Fig. 3.6 – right). Linear models only need to store their coefficients' values. Tree-based models must store a condition value and a feature identifier for each node. We assume that all condition and coefficient values require 64 bits and the number of features defines the number of bits necessary to represent them. The RR model needs 960 bits, the DT model 276kB, and the RF model 1.5MB. Overall, RF is expensive compared to the others. Although it does provide high accuracy, the predictions do not necessarily translate to a significant gain in scheduler efficiency, as observed later.

3.9 Scheduler Evaluation

This section materializes our insights derived in the exploratory part of this work. Specifically, we train our most accurate RR, DT, and RF models using only the features we identify as important in Section 3.7. Each model is then linked to a scheduler that receives a prediction matrix and derives a greedily optimal schedule to maximize performance. Using this scheduler framework, we run 200 workloads to completion on each of our 9 benchmark multicore systems identified in Section 3.4.1. We compare our schedulers against a greedy oracular scheduler that

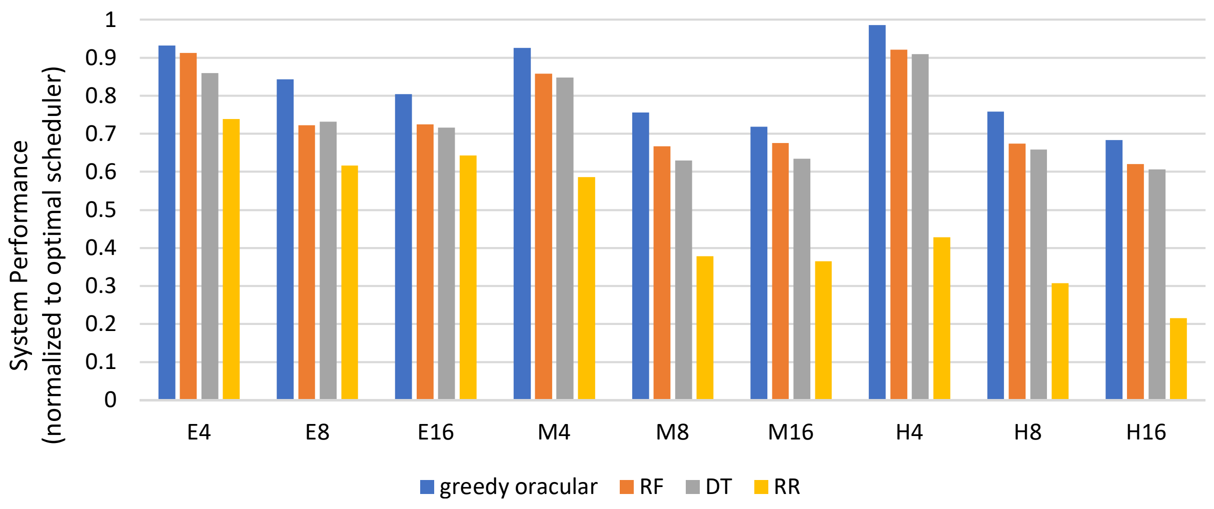


Figure 3.7. Scheduler performance comparison.

consistently makes 100% accurate performance “predictions” on every probe.

We no longer use the LOGO data splitting approach to measure the overall scheduler efficiency. Instead, we increase the size of the test set and reduce the size of the training set, by assigning four randomly chosen benchmarks (instead of one) to be our test set. While this could potentially result in reduced predictor quality in terms of MAE and STD, it enables a larger selection of previously unseen workloads allowing us to explore more realistic computation environments.

For this experiment, we randomly select input workloads that create a mix of 50% unseen (from test set) and 50% previously seen workloads (from train set), resembling a typical IaaS (Infrastructure-as-a-Service) environment (e.g., Amazon’s EC2) – some users use EC2 instances to run the same application every time (such as a web server), while others execute a more diverse mix of workloads (software development and testing).

3.9.1 Performance Evaluation on Heterogeneous-ISA Architectures

Figure 3.7 presents our results normalized to the average performance under the optimal scheduler that emits a schedule resulting in the maximum-achievable overall throughput. We first observe that the RF-based scheduler has an advantage compared to the other ML-based schedulers. However, the much cheaper DT-based scheduler scores very close, across all systems. Compared to RF, DT reports a 2.8% average performance reduction (6.2% max reduction), while it outperforms RF on the E8 system by 1.2%. We further observe that our RF scheduler is within 2.2-11.2% (7.5% on average) from an oracular scheduler, and DT within 1.6-16.8% (10% average).

Our RR-based scheduler shows significantly reduced performance. The reduced training set size affects our top RR predictor's accuracy significantly enough to move it past the error tolerance zones we identify in our ESE analysis. While the (test set) accuracy of all our models drops due to the reduced training data set, RR is affected the most, with its MAE doubling and STD increasing by almost 3x. For comparison, tree-based models only experience 6% MAE and STD degradation. Furthermore, tree-based prediction accuracy on the training set is slightly better (by 1%), which provides a significant advantage over RR in this experiment. RR remains roughly within the same levels of accuracy on the training set.

3.9.2 Performance Evaluation on Single-ISA Architectures

To demonstrate the adaptability of our schedulers to less complex systems, we also examine single-ISA heterogeneous systems. We repeat the EoS-based system selection presented in Section 3.4.1. However, this time we enforce our systems to only use Alpha cores. We must note that our predictors have not been re-trained between the two experiments. Figure 3.8 presents our results.

We first observe that, unlike with heterogeneous-ISA systems, the DT scheduler has an advantage over RF. The added complexity of the RF predictor does not appear to be as beneficial in the single-ISA case. We can also observe that our linear predictor now reports comparable efficiency as RF and DT (1.5-4.8%). This happens due to the reduced scheduling difficulty on single-ISA systems (higher EoS values). Our best scheduler (DT) performs within 0.2-12.5% across all systems compared to the oracular scheduler (4% on average).

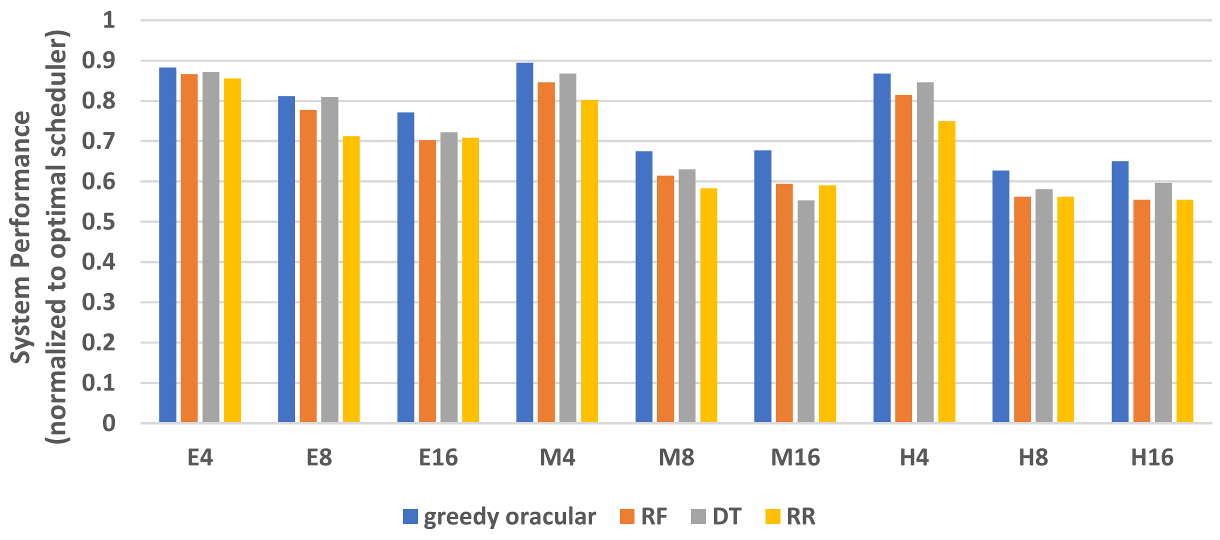


Figure 3.8. Scheduler performance comparison on Single-ISA heterogeneous systems.

3.9.3 Scheduler Sensitivity to Underlying System

From Figure 3.7, we observe that our tree-based (DT, RF) schedulers are affected by the transition from 4 to 8 cores, but their performance remains almost intact when we move from 8 to 16 cores. In comparison, the oracular scheduler is affected by both transitions ($4 \rightarrow 8$, $4 \rightarrow 16$), albeit with a smaller impact. These trends are also observed when our schedulers accompany single-ISA systems (Figure 3.8). Our results show that tree-based schedulers were able to mostly overcome (1) the increase in scheduling difficulty between 8 and 16-core systems, and (2) the variety of underlying ISAs.

3.10 Conclusions

With heterogeneity becoming more and more available in modern systems, predictive models enjoy increased attention from the scientific community for applications in scheduling, power management, resource management, resource allocation and more. In this work we first present an exhaustive exploration and analysis of the abilities of ML models to act as performance predictors. Our results reveal that accuracy metrics typically used in ML works do not necessarily translate to computer architecture applications such as scheduling in a predictable fashion.

Furthermore, we reverse-engineer trained ML models and conclude on which measurements are necessary for accurate predictions. We finally materialize our insights in three ML-based schedulers for heterogeneous-ISA systems and demonstrate their effectiveness on systems of varying scheduling difficulty.

Acknowledgements

Chapter 3, in part, is a reprint of the material as it appears in the proceedings of PMAM 2019. Prodromou, Andreas; Venkat, Ashish; Tullsen, Dean M., Deciphering Predictive Schedulers for Heterogeneous-ISA Multicore Architectures, 10th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM), held in conjunction with PPoPP 2019.

Chapter 3 further contains a partial reprint of the material as it appears in the proceedings of SAMOS 2019. Prodromou, Andreas; Venkat, Ashish; Tullsen, Dean M., Platform-Agnostic Learning-Based Scheduling, 19th International Conference on Embedded Computer Systems: Architecture MOdeling and Simulation (SAMOS). The dissertation author was the primary investigator and author of both these papers.

Chapter 4

Agon: A Scalable Competitive Scheduler

This chapter proposes a competitive scheduling approach, designed to scale to large heterogeneous multicore systems. This scheduler overcomes the challenges of (1) the high computation overhead of near-optimal schedulers, and (2) the error introduced by inaccurate performance predictions. We present Agon, a scheduler that employs a range of schedulers, from simple to very accurate, and learns which scheduler provides the right balance of accuracy and overhead for each scheduling interval.

4.1 Background

Recent years have seen an increase in interest in heterogeneous hardware, both in industry and in the research literature. Processors or systems that allow diverse execution engines maximize the likelihood that any particular thread or application finds an execution core most suited to its execution, resulting in more efficient (higher performance, lower energy) execution. This execution diversity can take the form of application-specific accelerators [19, 29, 102, 89, 52], as well as general purpose cores with diverse micro-architectures [62, 63] or even Instruction Set Architectures [32, 108, 106].

Harvesting the full potential of heterogeneity is not trivial, as it requires effectively assigning diverse tasks with constantly changing resource needs, to the best combination of hardware engines at any point in time. The larger the system and the greater the level of diversity, the more difficult it is to find the best solution. A large body of work can be found in the literature with proposals to address resource management in heterogeneous hardware [98, 31, 25, 78].

A variety of proposals [31, 24, 115, 9] focus on predicting the performance of each workload on each core of a heterogeneous system. This information can then be passed to the scheduler which will use it to generate a schedule (workload-to-core assignment). This research focuses on the scheduler itself. While several proposals have been successful in predicting runtime characteristics with high accuracy, many just assume the presence of a scheduling algorithm that can utilize their predictions. With small, or less heterogeneous, systems this assignment can be trivial, assuming the predictions are accurate. As systems get larger and more diverse, however, the workload-to-core assignment becomes computationally prohibitive, with the cost of the assignment algorithm potentially overwhelming the gains from a good schedule.

Thus, we find that in a complex system, a scheduler can fail (or perform suboptimally) in two ways – it can identify a poor schedule, or it can spend so much time computing the schedule that it significantly defers useful computation. The fact that scheduling needs can differ between cases is the foundation of this work. We propose *Agon*. Given the state of a system, Agon selects the appropriate scheduling algorithm (from a number of options), such that it maximizes overall system performance. For example, in cases where there is significant gain between the best schedule and a "good" schedule, it might employ a high-cost scheduler, but in another case where many of the potential schedules are near-optimal it will employ a low-cost scheduler. Thus, Agon is not a scheduler, but an arbiter between schedulers.

We design Agon as a trainable, neural network module. We further present and evaluate multiple configurations and neural network architectures that can be used. Some focused more on simplicity and efficiency, while others focus on improving Agon's prediction accuracy.

Agon has a dual-part architecture. Its novel front-end is trained to filter out performance prediction error (denoiser), much like neural network modules that remove noise from images. Its back-end is a typical classifier that receives performance predictions and outputs a choice of scheduling algorithm.

We evaluate Agon on a variety of underlying heterogeneous-ISA systems, multiple datasets, and multiple scheduling goals. Our results show that Agon is capable of selecting the optimal scheduler for each problem instance with accuracy up to 89%. We further find that simply due to avoiding scheduling overhead when possible, average system performance improves by $\sim 6\%$ compared to a system equipped with the best scheduler available, while at the same time we approach the performance of an oracular scheduler selector (99.1% of oracle performance).

We first present an overview of related work in Section 4.2. We then motivate our work via a detailed exploration of all the variables that can affect which scheduler should be used in Section 4.3. We discuss the details of Agon's architecture (Section 4.4), as well as the details of our dataset (Section 4.5) and evaluation methodology (Section 4.6). Section 4.7 presents and discusses our results, and finally Section 4.8 concludes the paper.

4.2 Related Work

Single-ISA heterogeneous architectures are proposed by Kumar, et al. [62], with processor manufacturers currently offering heterogeneous products [37, 53, 104]. *General-purpose* heterogeneous-ISA multicore architectures [32, 108, 106], include microarchitecturally heterogeneous cores that implement different ISAs and demonstrate significant added benefits compared to its single-ISA counterpart. A recent proposal by Venkat, et al. [106] further demonstrates how a heterogeneous-ISA CMP can be based on feature-diverse variations of a single ISA, significantly alleviating several barriers to adoption.

In this work we choose to evaluate our mechanisms on heterogeneous-ISA multicore architectures, primarily because such architectures significantly increase the difficulty of the scheduling problem. The three ISAs we use in this study are Thumb [66], Alpha[30], and x86-64[45]. Our proposal is however orthogonal to the underlying system and can be (re-trained and) used with any combination of cores, systems, and performance predictors.

Scheduling in the presence of heterogeneous hardware is typically a two-step process: First, some predictive mechanism attempts to predict how each workload will perform on each execution option (e.g. heterogeneous cores of the underlying system, independent systems such as heterogeneous servers, etc). The result of this first step looks like a 2D array of workload-core predictions, which is then fed to the second stage: the scheduler. The scheduler's task is to receive the prediction matrix and decide on the final job-to-core assignment that optimizes some execution aspect, such as overall performance, bandwidth, and/or power consumption.

Some schedulers (scheduling algorithms) suffer from scalability issues due to their algorithmic complexity. For example, the Hungarian algorithm [51] is the only known algorithm besides brute-force that guarantees finding an optimal assignment in polynomial ($O(N^3)$) time. As we show in this work, this complexity can be impractical for chips with as little as 8 cores. The scalability issues of scheduling algorithms have been studied extensively in the literature in various contexts, such as power management in dynamically heterogeneous multicores [112, 111, 15], real-time systems [16], and more [70].

We base our work on the scheduling algorithms and heuristics evaluated by Winter et al. [112]. The authors assume the decay of CMPs throughout their lifetime due to permanent errors, which lead to dynamic heterogeneity and evaluate how each scheduling algorithm performs. They also propose an iterative scheduling solution termed "local" scheduling. Unlike prior work, our goal is not to design a new scheduling heuristic, but to assume the existence of multiple scheduling algorithms and build a mechanism that selects the appropriate algorithm to be used.

Numerous proposals from the literature focus primarily on the first step of scheduling, workload-core performance prediction. Some simply assume the existence of a fast, optimal scheduler and focus entirely on improving prediction accuracy [87, 88, 115, 94], while others also attempt to alleviate its overhead. Two methods are commonly utilized to achieve that: Iterative methods perform job migrations upon scheduling intervals, slowly progressing towards a better schedule but likely sampling poor schedules along the way [94, 67, 100]. The second method logically divides large systems into smaller independent scheduling islands [23], allowing the use of more complex algorithms operating with limited-scope knowledge. Scheduling islands are a reasonable solution, especially for very large systems, however we show that limited knowledge can have a significant (in some cases) impact on overall system performance.

The Agon scheduler we propose is a classification neural network [12], with each class corresponding to the scheduling algorithm to be used. Classifiers are used extensively in a wide range of applications, from object recognition [61], to medical imaging (e.g. classifying tumors as benign or malignant [34]), to credit card fraud detection [35], and more. Agon explores various neural network architectures, such as basic Feed Forward Networks (FFN), convolutional layers, and convolutional autoencoder architectures used for denoising their input [109, 36]. Convolutions and convolutional autoencoders are typically used with image inputs. In this work we are able to utilize these architectures and operate successfully on 2D performance prediction matrices. In fact we find that denoising autoencoders can remove performance prediction error almost entirely when properly trained.

The overhead of neural network inference (running an input through the network and getting a prediction) has been and still is improving rapidly, due to the meteoric rise of inference accelerators over the last few years [52, 60, 103, 83].

4.3 Motivation

This work is motivated primarily by the fact that determining optimal or near-optimal schedules for large, heterogeneous systems is difficult to get right and computationally expensive. In this section we discuss the factors that make the scheduling problem hard: The computational overhead of the scheduler, the varying difficulty of hardware/workload combinations, the inaccuracy of performance predictions, and finally the scheduler's goal.

4.3.1 Scheduler Overhead

The scheduling problem we address in this work is a special case of the linear sum assignment problem [20]. The most efficient solution is the hungarian algorithm, which guaranteed to find (one of) the matrix permutations that generates the largest sum on its diagonal (this diagonal is a job-to-core assignment in our case). Many other heuristics have been developed, that do not guarantee the optimal, or in some cases even a good solution, but operate much faster than the hungarian solution.

The impact of a scheduler's algorithmic complexity has been demonstrated and discussed in prior work [112] in the context of dynamically heterogeneous architectures. These architectures begin as a homogeneous system, but become heterogeneous due to hardware failures during the system's lifetime. Besides that work however, the scheduling step is often simply assumed in performance prediction proposals. In other words, proposals that design performance predictors, assume the existence of a scheduler that will receive these predictions and generate the actual job-to-core assignment.

In reality, the hungarian scheduler, with $O(N^3)$ algorithmic complexity, cannot scale to large multicore systems. In fact, we find that in systems with as little as 8 cores, the overhead of hungarian scheduling is sometimes not covered by the returned performance benefit. On very large systems, hungarian scheduling can take as much as a full second to complete.

We measure scheduler overhead by creating random inputs of varying sizes (4-256 cores) and feeding them to each scheduling algorithm (implemented in Python). We repeat each experiment 10000 times to minimize random noise. Measurements were performed using an i7 processor running at 3GHz.

We present our results in Figure 4.1 (note the logarithmic Y axis). We compare 6 scheduling algorithms. Smart random is the only new scheduler that we develop in this work, while all others are described in prior work. Our measured overheads track each scheduler's algorithmic complexity.

As we discuss in the remainder of this section, we cannot safely assume that higher overhead will lead to better performance, or that the returned performance will be enough to cover the time spent in scheduling.

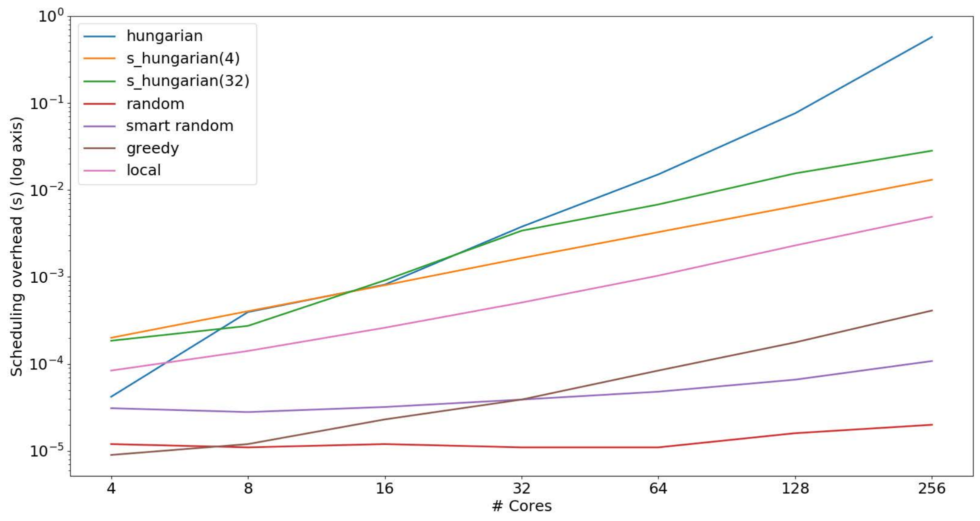


Figure 4.1. Overhead comparison of 6 scheduling algorithms (Logarithmic Y axis)

Scheduler Description

In this work we compare the following schedulers:

Hungarian Algorithm: Guarantees finding one (of many in case of ties) optimal solution to the linear assignment problem in $O(N^3)$ (Jonker et al. [51]).

Serial Hungarian: System is divided into smaller pods. In our exploration we evaluate pod sizes of 4 and 32. Each pod internally runs the hungarian algorithm and pods run serially. Algorithmic complexity remains $O(N^3)$, but N is the pod size.

Random: Given N workloads and assuming their order corresponds to the core each one executes, the random scheduler returns a shuffled list with algorithmic complexity $O(N)$.

Smart Random: If possible, this scheduler avoids assigning floating point workloads on Thumb ISA cores (which do not have FP hardware support). Besides this one rule, all other decisions are random. This scheduler has $O(N)$ complexity.

Local: Given some starting schedule (random at first), this algorithm randomly partitions the schedule into two halves and swaps workloads pairwise. If the new assignment is not beneficial compared to the previous one, swaps are reversed. This process is repeated N/2 times and spans N/2 consecutive scheduling intervals. It can be implemented with $O(N^2)$ complexity.

Greedy: Sorts workloads according to ILP and cores according to performance. Final schedule is a one-to-one assignment between the two lists. Can be implemented with $O(N\log N)$ complexity.

4.3.2 HW/SW heterogeneity and EoS

The first factor that can affect scheduler quality in our list is core heterogeneity. Rather than limit our study to a single instance of a heterogeneous system, we examine a large number of combinations taken from 600 unique heterogeneous cores and 3 ISAs, which is the extent of our dataset. Given a randomly selected N-core system, its member cores might be closer to each other in terms of performance or the difference could be quite significant. As inter core variance increases, we would expect that so does the difficulty of scheduling.

Software heterogeneity affects scheduling in a similar fashion as hardware heterogeneity: Software can be compute- or memory-bound, can include floating point or SIMD operations, can have varied register pressure, etc. Prior work has demonstrated that software, and even distinct phases within each program present “ISA affinity” [108] - an inherent preference to one ISA over others. This can be due to obvious factors such as the presence or lack (e.g. Thumb) of FP hardware, or more subtle factors such as register file size, addressing modes supported, code density, etc.

Combined, hardware and software heterogeneity can severely impact a scheduler’s ability to find a good job assignment. Prior work proposed an Ease-of-Scheduling (EoS) metric that quantifies the impact of these factors in respect to the scheduler’s outcome in a single number [87].

In this work we repeat the EoS experiment with our dataset. Due to the difference in scope between this and prior work, we modify the EoS experiment slightly: Instead of assuming a series of workloads, with a new one appearing as soon as another finishes, we only use single problem instances to the scheduling problem. A problem instance for an N-core system consists of an $N \times N$ performance prediction matrix - the input to the scheduler. For each N-core system we study, we run 20K problem instances and report the average performance across all of them. The 20K N-long workload sets are randomly chosen and do not change between systems. EoS relies on measuring the performance of a random scheduler. To minimize noise from random decisions, we randomly schedule each problem instance 10000 times.

Our results in Figure 4.2 show that for each system size (4/8/16 cores) scheduling difficulty can vary significantly, with some systems being quite easy to schedule, even for a random scheduler, while others can be quite challenging. Subfigures separate each system size and provide a non-overlaped view. In accordance to prior work, we use our EoS results to define the benchmark systems of this study: For each N value (4/8/16), we select the easiest (highest dot), hardest (lowest), lowest-performing (leftmost), and highest-performing (rightmost). In the remainder of this paper, benchmark systems are labeled as *E, H, LP, or HP* respectively, followed by the number of cores in the system. For example, *H4* is the label for the hardest-to-schedule 4-core system, and *LP8* is the lowest-performing, 8-core system.

4.3.3 Prediction Error

Agon assumes that some predictive unit generates the scheduler's input. As such we can safely assume that the scheduler, regardless of its complexity, will be faced with erroneous inputs. The quality of the predictor can significantly affect scheduling quality.

We measure the impact of prediction error on three schedulers: hungarian, serial hungarian and local. Random and greedy schedulers do not depend on performance prediction and are immune to prediction error. For fair evaluation, we assume the existence of a good predictor, since as prediction quality drops, all schedulers tend to converge to the performance of a random scheduler. Prior work has evaluated numerous such predictors, using a variety of different machine learning algorithms [87, 88]. We choose their Decision-Tree based predictor and assume it to be the generator of Agon's input matrix, due to its low overhead and high accuracy.

Figure 4.3 shows our results. We find that compared to oracular performance predictions,

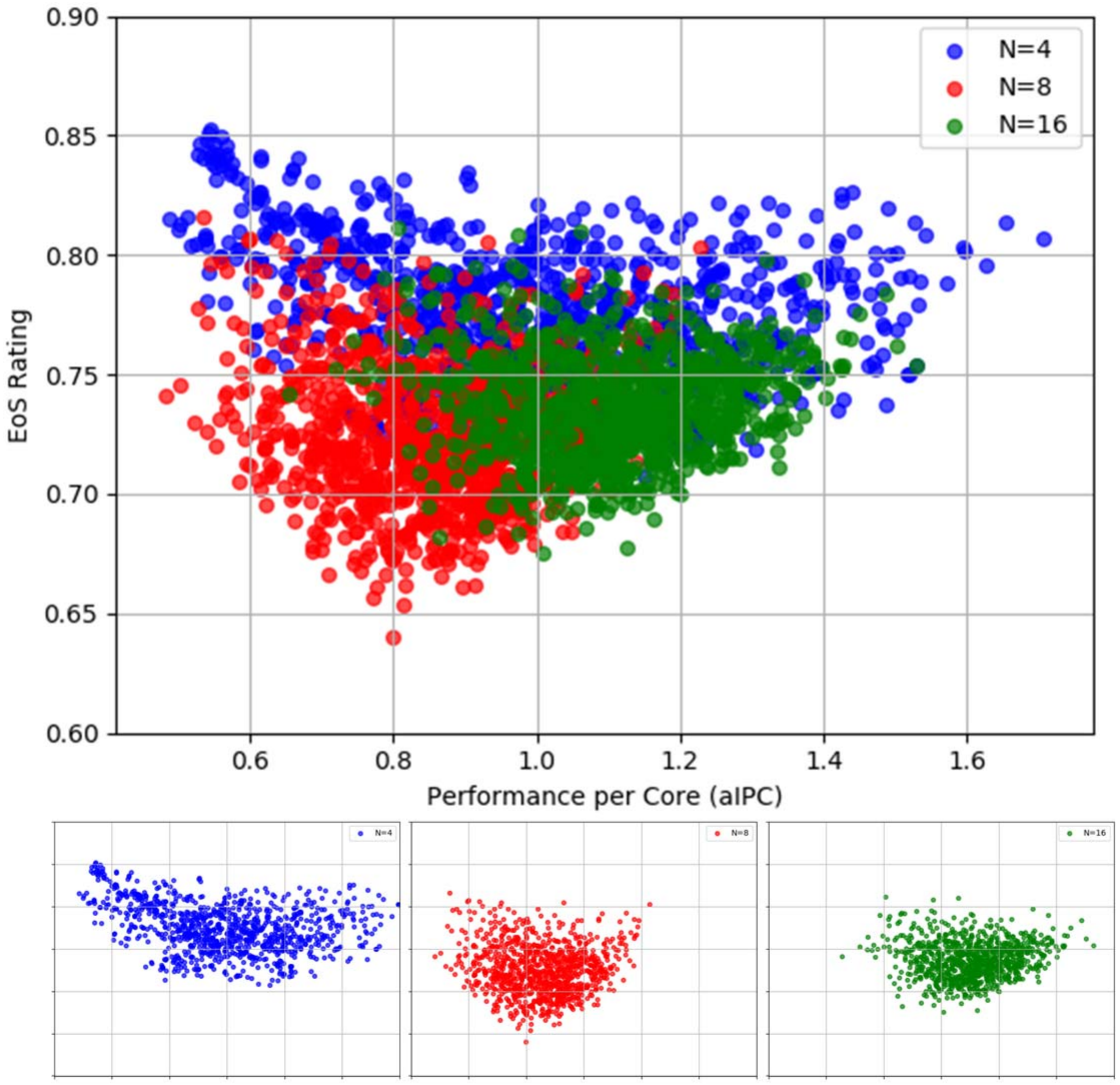


Figure 4.2. Modified Ease of Scheduling (EoS) Evaluation

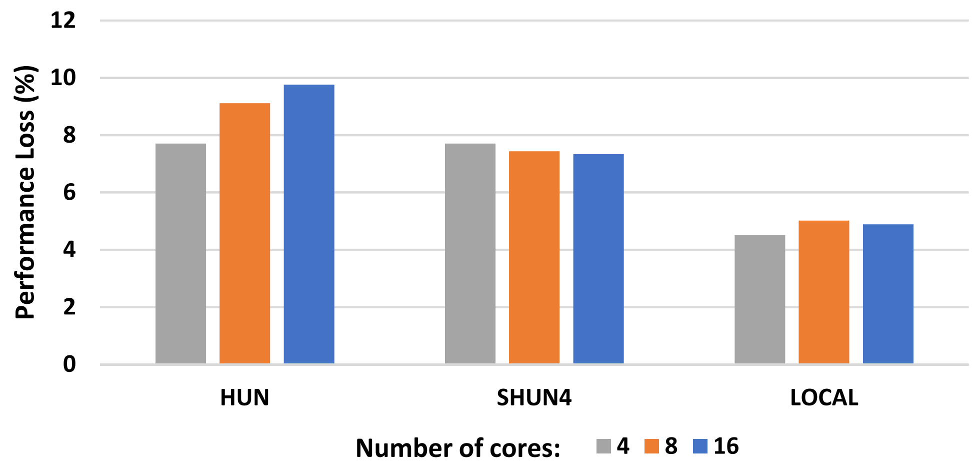


Figure 4.3. Impact of prediction error on three schedulers (Hungarian, serial hungarian with pod size=4, local). Predictor accuracy is modeled with a bell curve (mean error=0.22, standard deviation=0.35).

hungarian algorithms, both the originally proposed as well as the serial version, are affected between 7.2 – 9.8% on average by the prediction error of a **good** performance predictor. The local algorithm is affected at a lesser degree (4.1 – 4.9%), however, compared to the hungarian scheduler, local returns up to 21% and 17% lower performance on average for oracular and predicted inputs respectively.

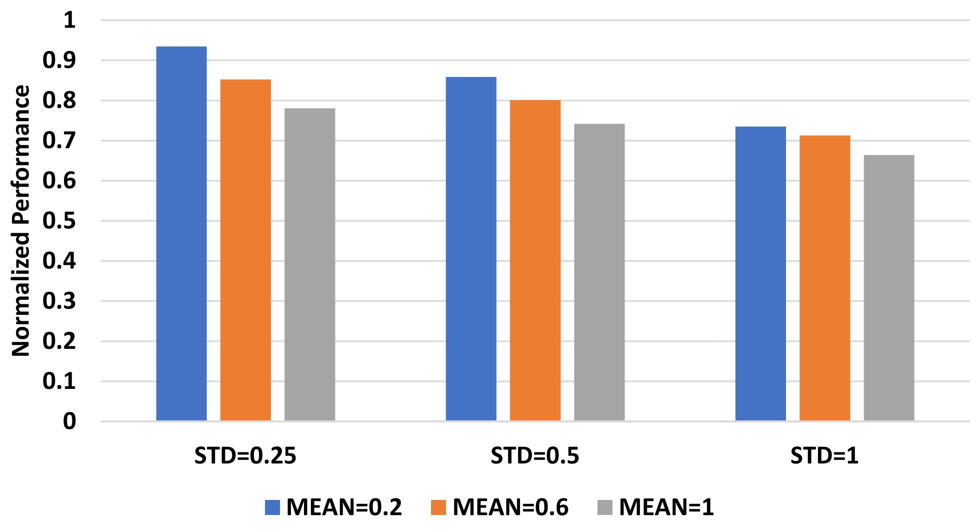


Figure 4.4. Normalized performance of schedules found using the Hungarian algorithm under performance predictors of varied quality. Predictors are emulated using the mean (MEAN) and standard deviation (STD) of their prediction error.

Figure 4.4 presents the impact of prediction error on a single instance of the scheduling problem (i.e. for one particular 8-core system and one set of 8 workloads). Both the system under test, as well as the set of workloads were randomly selected for this experiment. Performance (Y axis) is normalized to the performance of the best possible schedule, which is calculated using the hungarian algorithm fed with oracular (error-free) performance predictions. All nine schedules presented in the figure are generated using the hungarian algorithm and each bar corresponds to a different performance predictor that generates the scheduler's input. Each predictor is emulated using the mean and standard deviation of its prediction error. We minimize noise from random decisions by repeating the experiment 10000 times and reporting average performance of schedules. With the worst predictor generating inputs to the scheduler, the derived schedule performs 33% worse compared to the best possible schedule. We also observe that while mean error in predictions has a significant impact, so does the standard deviation of error our predictor exhibits.

4.3.4 Scheduling Goals

Some of the factors that can affect a scheduler's quality are also dependent on what we refer to as scheduling goals. For example, scheduling overhead (Figure 4.1), will have high impact on a scheduler that operates very frequently. A less frequent scheduler will spend less (percentage-wise) time on overheads and more on executing useful instructions.

Later in this study, we calculate the winner scheduler under a variety of scenarios, such as varied error at the classifier's input, varied system sizes, etc. The correct answer is of course dependent on what our scheduling goal is.

In this work, we explore 4 different scheduling goals: The **FIRST100** goal, defines the winner scheduler to be the one that will first execute 100M Alpha instructions (or equivalent work on another ISA). Similarly, our **FIRST400** goal selects the scheduler that will first execute 400M Alpha instructions. **MOST50** selects the scheduler that will execute the highest number of instructions within 50M cycles and finally, **MOST200** allows for 200M cycles.

In calculating the winner scheduler, we take into account each scheduler's overhead. For example, if the hungarian scheduler needs 10M cycles to execute and the goal is set to MOST50, this leaves the scheduler with only 40M cycles to execute useful instructions. Thus, each goal is impacted by the computational overhead a bit differently.

4.4 Architecture

Agon is a "competitive" scheduler. Conceptually, a number of schedulers (scheduling algorithms) compete to be the one to solve the assignment problem. Agon acts as the arbiter which decides the winner in each instance. In a realistic execution scenario, available input information will be an $N \times N$ performance matrix, representing the (predicted) performance of each of the N in-flight workloads on each of the N cores.

A performance predictor can be responsible for generating the performance matrix. Proposals from the literature that tackle this particular problem are presented and discussed in Section 4.2. During evaluation, we also examine the impact of oracularly obtained performance matrices, however we cannot realistically expect the presence of 100% accurate predictions during runtime, unless we operate in a highly predictable environment.

Figure 4.5 shows a high-level overview of our mechanism's architecture: Agon acts as the control signal of a K-way multiplexer, where K is the number of available schedulers. Internally, Agon runs the input matrix through its (optional) denoising front-end, followed by its classification back-end . A series of hidden layers in both modules process the information. The classifier's output layer consists of K nodes. With the output nodes using softmax activation [71], Agon's output is the probability of each scheduler to be the most effective one. We then select the scheduler with the highest probability as the winner.

At its heart, Agon is a classifier, which receives a performance matrix as input and selects one (of six in this study) available schedulers. We explore a variety of possible architectures for Agon, with varied complexity, classification accuracy, and system performance. We describe

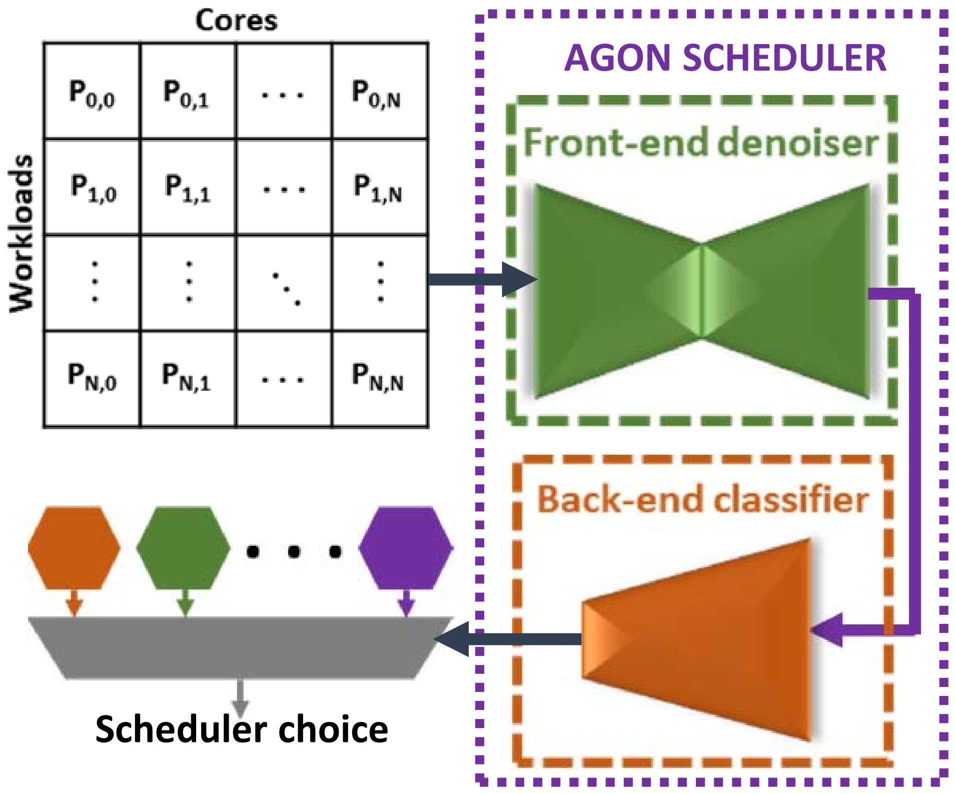

Cores
Workloads
$P_{0,0}$ $P_{0,1}$ ... $P_{0,N}$
$P_{1,0}$ $P_{1,1}$ ... $P_{1,N}$
$P_{N,0}$ $P_{N,1}$ ... $P_{N,N}$
AGON SCHEDULER
Front-end denoiser
Back-end classifier
Scheduler choice

Figure 4.5. High-Level overview of Agon Architecture: A performance prediction matrix is flows through Agon's front- and back-end modules and finally guides a scheduler selector.

these architectures in the remainder of this section.

4.4.1 Classification Back-End

This is the main control center of Agon, where the final decision is made. We explore two neural network architecture options: A simple, feed forward architecture (labeled “deep” in the remainder of this work), as well as a convolutional classification architecture (“cnn”). Both architectures result in a softmax output layer with 6 nodes, since we are training it for 6-way classification.

As a classifier, the operation of Agon’s back-end is quite straightforward: The returned class is the one with the highest (softmax) probability. The input layer is internally ensured to be compatible with the selection of denoiser (described in Section 4.4.2 - via the use of pseudolayers such as “Flatten” and “Reshape”). For example, a convolutional denoiser will output a 2D matrix, while the simple denoiser will output a flattened 1D list. Similarly, a feed-forward classifier will expect a flat input, while a convolutional one will expect a 2D input.

Table 4.1. Layer breakdown: Deep FFN classifier

Layer ID	Layer Type	# Nodes	Activation
1	Dense	128	ReLU
2	Dense	32	ReLU
3	Dense	6	softmax

Tables 4.1 and 4.2 present more details about the architectures we explore. We have explored various architectural options and selected the two most efficient for our study.

4.4.2 Denoising Front-End

We propose an autoencoder-based front-end, responsible for removing prediction error (if any) from the input matrix, which is produced by some performance predictor. This idea is inspired by neural-network-based noise-reduction techniques for images. The parallels between our use case and noisy images is apparent: Both are 2D arrays of values and prediction error

Table 4.2. Layer breakdown: Deep convolutional classifier

CNN section of classifier		
Layer Type	**# Filters**	**Kernel Size**
Conv2D	64	(4,4)
MaxPooling2D	NA	(2,2)
Flatten	*Pseudolayer: Flattens output matrix*	

FFN section of classifier		
Layer Type	**# Nodes**	**Activation**
Dense	64	ReLU
Dense	32	ReLU
Dense	6	softmax

can definitely be considered as noise. Given that autoencoders are very successful in denoising images, we expect it to also work well in our example. To the best of our knowledge, this is the first neural network based proposal that attempts to reduce prediction error after predictions are made.

Autoencoders are typically trained to output their input, after its dimensionality has been internally reduced. In other words, during training, labels (the correct answers) are identical to the input. Denoising autoencoders on the other hand receive some realistic input during training (e.g. a noisy image). They are trained to output the oracular (i.e. image without noise) data at their outputs. We adapt this methodology and train our denoising frontends with realistic, non-oracular predictions from our performance predictor module, and use our oracular dataset as labels. Our datasets are described in more detail in Section 4.5.

Similarly to the back-end classifier, we explore two denoiser architectures for Agon: A simple, feed forward architecture, and a convolutional architecture. The feed forward architecture, labeled “simple” for the remainder of this work, consists of three layers. For 8-core systems, these layers would have 64 nodes at the input and output layers, and 32 nodes in the one hidden layer. We experimented with deeper architectures with more hidden layers, however this simple one was evaluated to be the most successful one.

Convolutional neural network architectures are typically used along with images, due to

Table 4.3. Layer breakdown: Simple FFN denoiser

Layer ID	Layer Type	# Nodes	Activation
1	Dense	32	ReLU
2	Dense	64	ReLU

their ability to recognize shapes, forms, and other elements found in images (information that is typically lost when the 2D image is flattened into a 1D array of pixel values). Convolutional autoencoders are particularly successful at reducing noise from images.

A (predicted) performance matrix might not have any meaning as an image, however, we claim that a convolutional "deconstruction" can still be meaningful: Assume we have a system of 8 cores (8×8 input matrix) and a convolutional 4×4 window that *slides in 2 dimensions* over the input. Also assume that the input matrix is a greyscale image (higher values are darker). The (sliding) convolutional window will look at the input in groups of 4x4. Each 4x4 sub-matrix might include a shape – for example a dark diagonal, a straight line, a shape similar to a block in the game of tetris, a diamond, etc. All these "shapes" are meaningful to the denoiser and through training over many such subfigures it can learn to darken or whiten (increase or decrease the predicted performance) of certain entries of these sub-matrices such that the shape is consistent with its training. Because of the 4x4 sliding window, each entry of the matrix will be examined 16 times, in 16 different neighboring sum-matrices, allowing the denoiser to have even higher confidence in its changes.

Besides the two denoiser architectures, we also evaluate Agon without a denoiser. In conclusion, the three available denoiser options for Agon are: none, simple, and cnn. Tables 4.3 and 4.4 present more details for our proposed architectures.

We utilize Tensorflow's Keras API and design each back-end classification module with a non-trainable input layer that reshapes the input to whatever shape is expected in order to create standalone front- and back-end modules that can be easily combined.

Table 4.4. Layer breakdown: Deep convolutional denoiser. All Conv2D layers use the ReLU activation function. Where it applies, padding is set to "same"

Layer Type	# Filters	Kernel Size	
Conv2D	32	(3,3)	Encoder
MaxPooling2D	NA	(2,2)	
Conv2D	32	(3,3)	
MaxPooling2D	NA	(2,2)	
Conv2D	32	(3,3)	Decoder
UpSampling2D	NA	(2,2)	
Conv2D	32	(3,3)	
UpSampling2D	NA	(2,2)	
Conv2D	1	(3,3)	

4.4.3 Front-End, Back-End Synergy

Agon's two-part decoupled architecture generates some opportunities for synergy between the front and back-ends. An individually trained front-end denoiser with high efficiency, can be used in conjunction with a classifier that has been (also individually) trained using *oracular* data. Assuming successful denoising, there is no reason to train a classifier with data that comes straight from a predictor. On the other hand, if the classifier expects stellar, error-free predictions, it could fail in the presence of even small error, if any is left from the denoiser.

We can also train the denoiser and classifier together as one linked unit, in which case something interesting happens: The front-end no longer acts as a denoiser. Instead, it will learn to transform the erroneous input into a new matrix (which may or may not be error-free, may or may not look similar to the actual input), but generated with the back-end in mind. In other words, the front-end will *generate a classifier input from the real input, such that the classifier's accuracy increases.* Our evaluation shows that this indeed happens, with linked Agon models outperforming even classifiers trained using oracular data.

4.5 Dataset Generation

We start with our in-house, simulation-based dataset of 72 workloads, each one simulated in isolation on 600 different cores. These cores are divided into three groups of 200 (heterogeneous) cores each, one group for each of the three ISAs of our study: Arm's Thumb, Alpha, and Intel's x86-64. The workloads are benchmark phases of 100 million *Alpha* instructions each, identified using Simpoint, from the SPEC2006 benchmark suite (10 benchmarks used). Our in-house, LLVM-based compiler infrastructure allows us to map code regions (phases) to their equivalent code regions in compiled executables for a different ISA. The dataset contains workload profiling results (Gem5, McPat), as well as microarchitectural descriptions of each core. Overall, this dataset contains 43200 entries. Each entry contains 30 features that characterize the workload (e.g cache access stats, fetch/issue rates, performance in aIPC (Alpha-IPC), etc), and 18 features that describe the core's architecture (e.g. in-order or out-of-order, cache hierarchy, SIMD support, number of functional units, registers, etc).

4.5.1 Generation of Agon's dataset

Agon requires a different dataset for its training, as its input is an $N \times N$ matrix of predicted performance for each workload-core pair (N is the number of cores of the multicore system under test).

We first generate a list of 3×1000 multicore systems, 1000 systems for each N (4/8/16). The cores of each system are chosen randomly out of the 600 cores in our dataset (1.6×10^{22} possible 8-core system combinations). We force our system generator to ensure each system includes at least one core from each of the three ISAs, in order to ensure significantly heterogeneous(-ISA) systems that consequently make scheduling more challenging. We then generate a list of 20K workload combinations in a similar way, by randomly picking from our list of 72 workloads (7.2×10^{14} possible 8-workload combinations). We generate three lists of workload combinations, one for each size of N in our study. All 20k workload selections for each value of N can run on each of the 1000 systems of the same size, leaving us with a dataset of 60M entries (20M for each N).

Each entry (row) in our dataset is populated with the *measured (profiled via simulation)* performances of the workload-core pairs. For example, on 8-core, 8-workload systems, each dataset entry will have 64 performance measurements. This becomes our *oracular* dataset.

We use the oracular dataset as a comparison point later in our explorations, but it is also necessary for calculating the correct answer (label, scheduler to be used), for each entry of our dataset: Using oracularly obtained information, we execute each scheduler using the same six scheduler implementations used to generate Figure 4.1 and we receive one schedule from each one. For each of the 6 produced schedules (which may or may not be different from each other), we calculate its *effective aIPC*, which takes into account the scheduler's overhead. Finally, whichever scheduler produces the highest effective aIPC is considered the correct answer. Even in cases where we don't use an oracular, error-free input matrix, Agon is required to output the labels we calculate using oracular data since, regardless of prediction error, that still is the correct answer.

We generate a total of 4 datasets for this study: The first is oracular which we just described. We also generate two predictor-based datasets, using trained predictors from prior work: One Decision Tree (DT) predictor, which has been trained with 6/10 benchmarks, while the remaining 4 are reserved for testing as previously-unseen data. We term this dataset "multiLOGO DT", due to the multi-LOGO data splitting process we used. This process is similar to the methodology used in prior work [87]. The third dataset also uses a DT predictor, this time however, we train it with the entirety of our dataset to reflect a predictor in a system that executes the same workloads over and over again. In our study we use it mostly as a comparison point, while we focus on getting good results on the multiLOGO-DT. We name this third dataset "fully-trained DT".

Our last dataset is called the "statistical" dataset. Our DT performance predictors' error can be modeled as a bell curve, with a mean value and standard deviation. Specifically, our

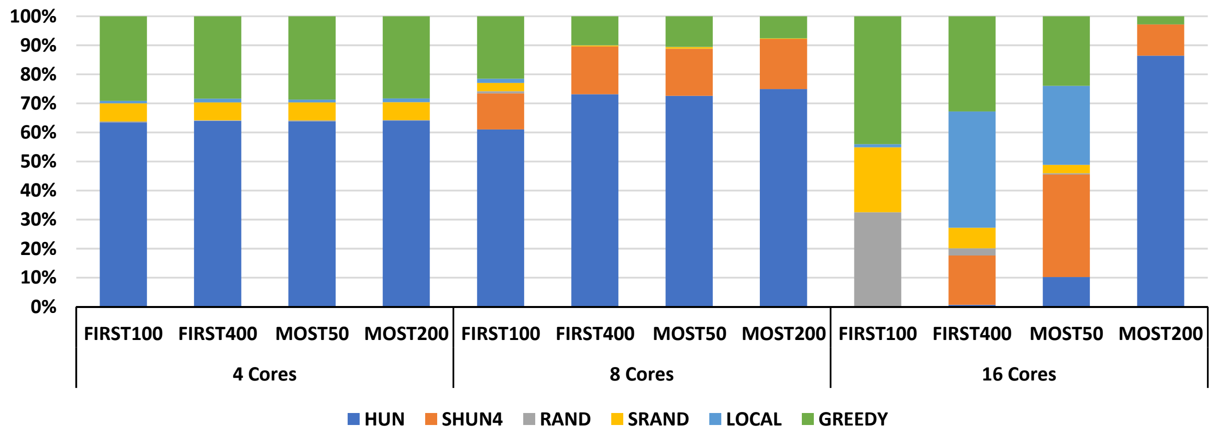


Figure 4.6. Breakdown of preferred schedulers for each entry in our dataset.

multiLOGO DT performance predictor reports $mean = 0.25(aIPC)$ and $std = 0.35$. We generate the statistical dataset by randomly drawing a prediction error value from the bell-curve that characterizes our predictors and simply applying it on the oracular value we get from the oracular dataset. Statistical error injection makes Agon's job artificially more difficult. Any real predictor is likely to make errors with some consistency or pattern that Agon could potentially learn.

4.5.2 Dataset Imbalance

Figure 4.6 presents a breakdown of the labels in our dataset, which correspond to the optimal scheduler choice of each entry in our dataset. The x-axis of the figure presents a variety of *scheduling goals*, as discussed in Section 4.3.4. *HUN* and *SHUN4* are shorthands for the hungarian and serial hungarian (with pod size of 4) schedulers respectively.

Our results show that our dataset is imbalanced in favor of the hungarian algorithm, which dominates most bars in the figure. Dataset imbalance is a challenge in training a good classifier. On a 4-core system for example, the hungarian scheduler is preferred for the vast majority of cases, due to its low overhead (since we only have 4 cores) and the fact that it tends to lead to the highest performance. As systems grow in size, we see a trend to move away from expensive hungarian-based schedulers and more towards random and greedy (i.e. ultra-low cost) schedulers. However, in systems where we run the scheduler less frequently (such as *16* *cores – MOST200*) hungarian again is important, because relative overhead is reduced and the importance of a good schedule is increased because it runs longer. However, some of our large systems with reduced scheduling frequency, rarely use hungarian.

This imbalance makes it easy for classifiers to learn to always pick one class. If hungarian is the correct answer 90% of the time, always picking it guarantees a 90% prediction accuracy. This classifier is of course not useful and could simply be removed from the scheduling process. Dealing with imbalanced datasets is actively pursued in the literature, since many crucial problems require good performance on edge cases (e.g. credit card fraud, self-driving cars). These are also problem categories where mispredicting corner cases can lead to catastrophic errors. A few options found in the literature are: Upsampling of corner cases, downsampling of popular cases, algorithmic dataset-driven generation of new (random) entries, and using class weights during training [59]. We find upsampling to work best in our study: We clone the entries of under-represented classes until the count of each class in our dataset is roughly even. Due to upsampling, the classifier will no longer blindly choose one option. On the other hand however, we have to be careful when dividing our dataset into training and testing subsets, since we don't want cloned entries to appear in both sets. Statistical error injection is immune to this issue since the applied error will differ between cloned entries.

4.6 Evaluation Methodology

We evaluate Agon on 4 datasets, 12 heterogeneous-ISA multicore systems, 4 scheduling goals, 6 different schedulers and 6 proposed neural network architectures. This section presents our experimental framework and training methodology.

4.6.1 Experimental Framework

The majority of our framework is developed in Python, using the Tensorflow framework for neural network training and evaluation. More specifically, we use the Keras API with a Tensorflow backend, due to its simplicity. Our front-end denoisers, back-end classifiers and complete Agon models are implemented using the *Sequential* model API and we use the standard Keras layers (Dense, Conv2D, etc).

During our evaluation, we assume that workloads appear in groups of N and each instance is independent. In other words, we assume an 8-core (for example) system that is presented with 8 workloads as soon as the previous batch of 8 workloads finishes. Each batch of 8 workloads represents a new scheduling problem instance (i.e. new predicted performance matrix, new prediction from Agon). In the majority of our results, we report the average performance after all batches finish executing.

We use *effective aIPC* as our performance metric in this work. We use Alpha Instructions Per Cycle (aIPC) to fairly compare the progress of executables of different ISAs, as proposed in prior work [87]. Effective aIPC represents a system's true performance once scheduling overhead is applied, as discussed earlier in Section 4.3.4.

We do not take into account Agon's overhead, even though it is definitely not negligible. Progress in neural network inference hardware accelerators leads to ever increasing inference performance [52, 60, 103, 83]. Even though Agon (or at least its more expensive versions with CNN denoisers etc) might not be viable today, we fully expect its overhead to be negligible in the future.

4.6.2 Agon Training Methodology

We train a different, independent Agon model for each underlying system. We utilize the *EarlyStopping* callback from Keras to avoid overfitting: This callback will automatically stop the training process once accuracy over the validation test ceases to improve for a number of consecutive training epochs.

We divide our dataset into three sets: 85% of our data is assigned to the training set, 5% for our validation set and the remaining 15% forms our test set. We use the training and validation sets during training. The validation set does not train the models (i.e. does not update weights), it simply exists for quick evaluation during training. The third set remains completely unseen by our models until they are finished training. We then use the test set to evaluate each model and report our results.

Agon's backend classifier uses "sparse categorical crossentropy" as its loss function. This function is implemented in both Tensorflow and Keras and is typically the loss function of choice for multi-class classification problems such as ours with a softmax output layer. By extension, Agon models that link a denoiser and a classifier for simultaneous training also use this loss function since the end goal is accurate classification. When we train front-end denoisers in isolation, we use "Mean Absolute Error" as the loss function, since we want the autoencoder's output to be as close as possible to the oracularly obtained performance matrices. Finally, we use the Adam [58] optimizer for all our training.

We focus our evaluation on 8-core systems and the "FIRST100" scheduling goal to narrow the scope of the work to a size that allows us to come to conclusions without harming the generality of our proposal. We choose this particular combination due to the healthier distribution of scheduling labels, as shown in Figure 4.6.

4.7 Results

4.7.1 Classifier Accuracy

Figure 4.7 presents our classifier's prediction accuracy, for a backend-only configuration (i.e. without a denoiser frontend). Prediction accuracy reveals our classifier's ability to pick the correct scheduling class. In other words, it is the ratio of correct scheduler predictions. For comparison, a random arbiter would have 16.6% $(1/6)$ accuracy. Specifically, we present a two-fold comparison: For each of our 4 benchmark systems we compare the classifier's accuracy under our oracular (unrealistic), multilogo (realistic), and statistical (artificially more challenging) datasets. We further compare the two classification architectures we propose for Agon. The deep feed forward architecture is presented on the left half of the figure, while the convolutional classifier architecture is shown on the right half.

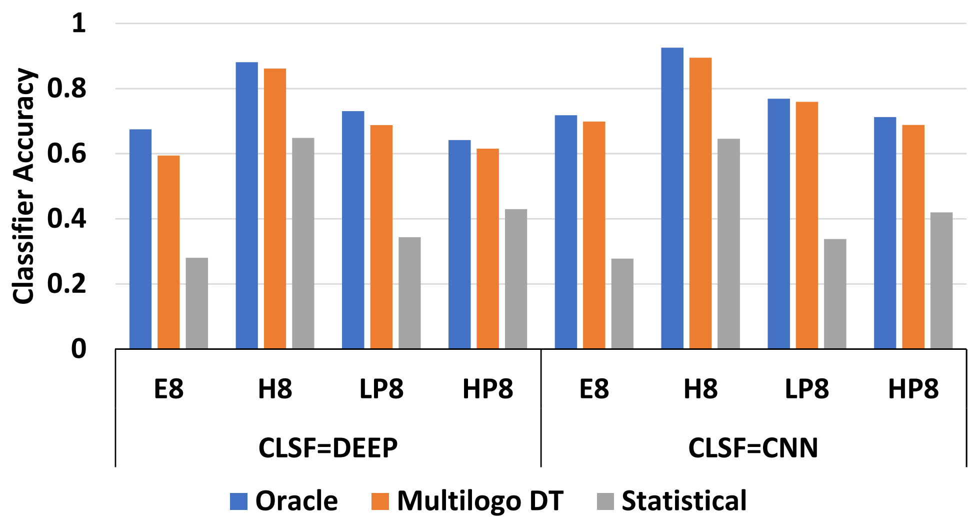


Figure 4.7. Classifier Prediction Accuracy. DEEP Vs CNN classifier architecture, oracle Vs multilogo Vs statistical datasets.

We first observe that accuracy on the realistic dataset is comparable with the unrealistic oracular. Specifically, the difference in prediction accuracy ranges between 8% (E8, DEEP) in the worst case to 1% (LP8, CNN), practically within noise levels. This result hints that our classifiers are able to reach the predictability limits of our problem. In other words, the missclassified cases are roughly those that would be missclassified even with oracular information. Due to the good quality of our multilogo predictor and its error consistency the classifier can still work efficiently with the erroneous input. Results over the statistical dataset show that prediction error inconsistency can severely impact classification quality.

Our results further show the convolutional classifier outperforming the simpler deep architecture. Specifically, CNN outperforms DEEP by 3.7-7.1% (oracular), 3.3-10.4% (multilogo), across all systems. The two architectures perform almost identically on the statistical dataset, again due to the unpredictability of the input.

The underlying heterogeneous-ISA multicore system is shown to have an impact on predictability as well. In fact, our results show an inverse correlation between a system's EoS (difficulty of a scheduler to find a good schedule), and the system's difficulty level in predicting the most efficient scheduler. Using a convolutional classifier on the multilogo dataset, Agon reports accuracy up to 89% for the most difficult, lowest-EoS system (H8). The other three systems report accuracy between 68-76%. The deep architecture reports lower accuracy, but still follows the same trend.

Prediction accuracy is not necessarily representative of the resulting performance we should expect. For example, if we misspredict one problem instance of the E8 system, we should still expect the (wrong – not most efficient for this instance) scheduler to produce a good schedule, due to the increased odds of success (high EoS). We present a performance comparison later in this section.

4.7.2 Denoiser Accuracy

Agon's denoiser (when individually trained) attempts to minimize Mean Absolute Error (MAE) and bring the output layer's values as close to ground truth as possible. We present our denoiser MAE results in Figure 4.8 (lower is better). On its own, this experiment does not provide any insight on how Agon improves with the addition of a denoiser. However, our results demonstrate that the denoising problem is in fact solvable, allowing computer architects to attempt denoising on other problems where hardware predictions are utilized.

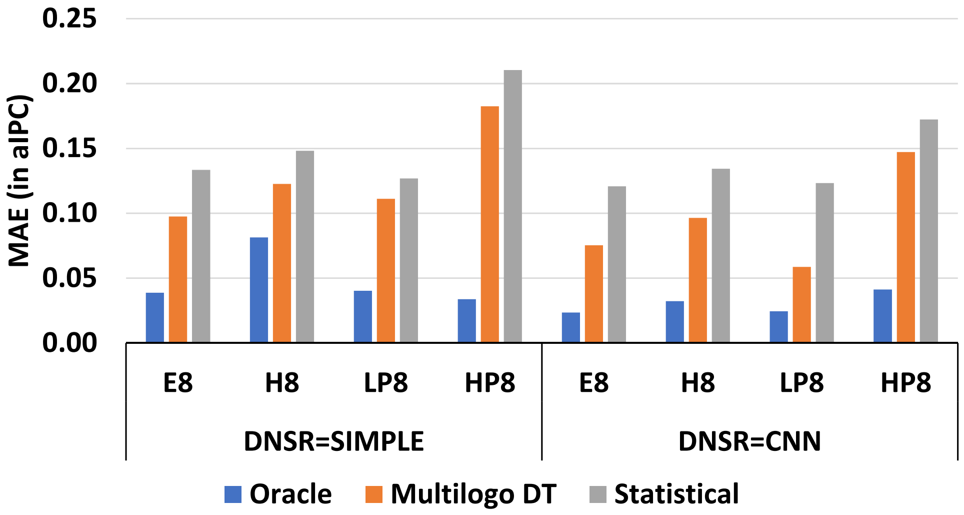


Figure 4.8. Denoiser loss in MAE (lower is better). SIMPLE Vs CNN classifier architecture, oracle Vs multilogo Vs statistical datasets.

Similarly to prior results, we present a comparison between three datasets. Denoising the oracular dataset represents an interesting case, which corresponds to typical autoencoder operation, where the output is trained to match the input, while internally, the dimensionality of the input reduces in the network's hidden layers – in our "simple" architecture, 64 input values reduce to 32 before fanning out to 64 output values. Results on multilogo and statistical, require true denoising, since we ask the frontend module to output oracular values, while receiving erroneous inputs.

In our use case, MAE is measured in aIPC, which in our dataset can be as high as 4.5 aIPC. Agon's frontend successfuly restores oracular inputs with absolute error below 0.05 in all cases but one (0.08 MAE), demonstrating that while dimensionality reduces, there remains enough information in the hidden layers to accurately characterize the entire input.

When dealing with multilogo erroneous inputs, denoiser error grows significantly compared to oracular, remaining however between 0.06-0.18 aIPC (0.11 aIPC on average). Finally, the statistical dataset, which is our most challenging case, exceeds 0.2 MAE in only one case. Overall, MAE on statistical input ranges between 0.12 and 0.21 (0.14 on average) aIPC.

4.7.3 Combined Agon Prediction Accuracy

Figure 4.9 presents a denoiser's effect in prediction accuracy, for the two classifier architectures we study and between multilogo and oracular datasets. Each bar cluster includes three data points: *NONE* corresponds to a backend-only Agon, *SIMPLE* and *CNN* correspond to the two denoiser architectures. For each case, the classifier's architecture is displayed under each group of benchmark multicore systems.

We observe that in almost all cases, the addition of a denoiser can improve the prediction accuracy compared to a "naked" backend-only Agon. In some cases, the *CNN-CNN* Agon neural network becomes too large for the problem and training halts (weights become NaN or zero), resulting in the few cases of chance-level prediction accuracy (16-20%) seen in the figure.

Interestingly, the addition of a frontend module improves prediction accuracy even in

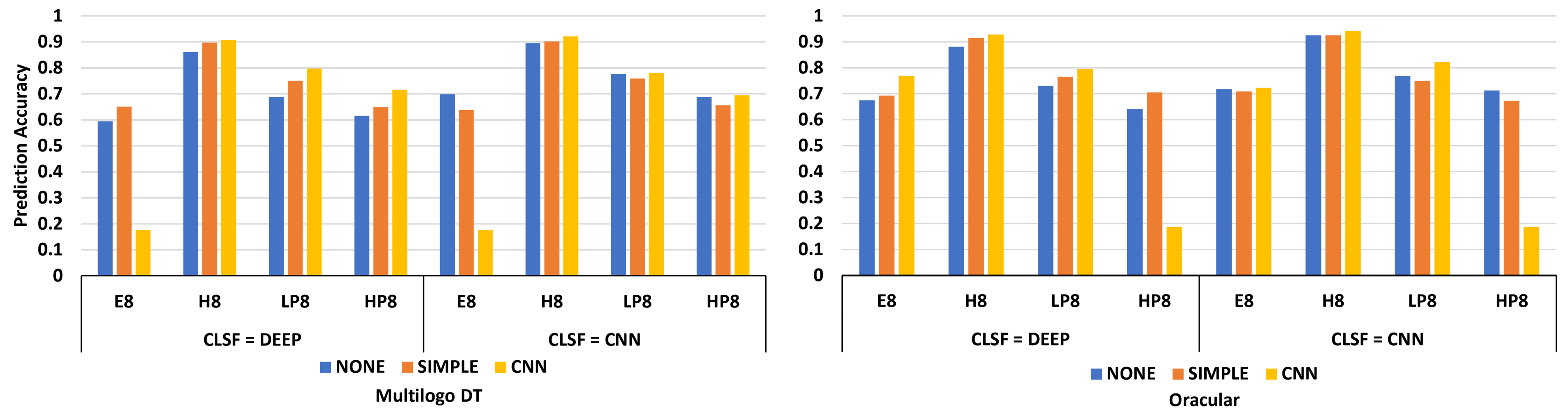


Figure 4.9. Impact of frontend on Agon's prediction accuracy on multilogo and oracular datasets. Agon's denoiser and classifier are trained together as one unit.

oracularly trained, oracularly-fed classifier models. Our results verify the expectation of a generative (rather than denoising) frontend. Agon's frontend manipulates the (oracular in this case) input in such a way as to improve the linked classifier's accuracy. We observe the same behavior in the majority of cases of the oracular part of our results.

Our observations using the oracular dataset present some insights as to the behavior of Agon's two parts. On the other hand, results on the more realistic multilogo dataset present the real-world impact of our proposed architecture.

The addition of a simple denoiser is observed to be beneficial in many cases of our experiment. Paired with a deep classifier, a simple denoiser boosts prediction accuracy by 3.6-6.3%. Replacing the simple denoiser with the much more complex convolutional architecture (still using a deep classifier) further increases accuracy to 4.5-11% with the exception of the failed case.

Pairing a denoiser with a convolutional classification backend presents new behavior. First, adding a simple denoiser is no longer beneficial for prediction accuracy. The naked classifier's accuracy is negatively impacted by 0.7-6% (the H8 case is improved by 0.7%). Our results hint that our convolutional classifier is already complex enough to handle error at its inputs. It also appears that even though the CNN classifier can render a simple denoiser unnecessary, it can still (slightly) benefit from a more complex convolutional denoiser (0.5-2.6% accuracy improvement).

4.7.4 Agon Performance

We finally present a performance comparison of Agon in Figure 4.10. Specifically, we compare the six Agon configurations we propose in this work, against six "always" scheduling strategies. We refer to a strategy as "always", when there is no competition between scheduling algorithms. For example, the always-RAND bar (4^{th} bar from the left in each group) represents a system equipped with only a random scheduler. The first bar in each group (labeled ORACULAR) represents the best case scenario, where we oracularly pick the optimal scheduler for each

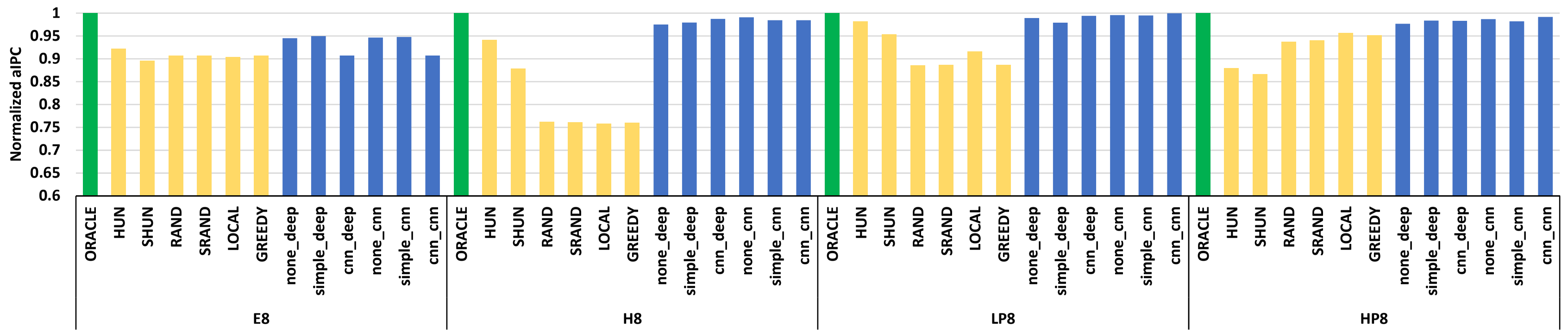


Figure 4.10. Performance comparison of 12 scheduling strategies, normalized to oracular knowledge of optimal scheduling algorithm for each problem instance. The first 6 strategies are "always" strategies, where there is only one scheduler option. Last 6 bars are Agon architectures.

problem instance. Agon configurations are labeled with *[frontend]_[backend]* keywords.

For each system, we run our 20K problem instances through each of the 13 strategies and report average system performance in aIPC. We normalize our results to the ORACULAR performance. We only present results using our realistic multilogo dataset for this experiment.

Starting from the E8 system, Agon boosts performance by $\sim 4\%$ compared to the most efficient always strategy (always-hungarian). Specifically, the best Agon configuration performs within 5% of an oracular selector. The two cnn_* Agon configurations are outperformed by most always strategies for this system, however in the remaining 3 systems, all versions of Agon outperform the best always strategy.

In very difficult (H8) systems, we see the always-HUN and always-SHUN4 (Serial Hungarian, pod size=4) cases performing much better than the rest of the always strategies. All versions of Agon outperform the best always strategy, with the *none_cnn* configuration reporting 99% of the oracular performance (compared to 94% for always-HUN).

Our results on the LP8 system show that always-HUN is comparable in performance to the best Agon configuration. The lowest-performing 8 core system consists of a combination of weak cores and, as a result, the majority of schedules for each problem instance will lead to roughly the same low performance. Consequently, focusing on selecting the best scheduler for each case appears to be unnecessary, even though we still measure Agon to perform within 1% of oracle.

In contrast to LP8 however, HP8 presents a different image. First, we observe hungarian strategies failing. This is due to the increased cost of each cycle spent in scheduling, as this system can get the maximum amount of work done in one clock cycle compared to all other systems in our study. This is also the reason we observe the low-overhead schedulers performing well. Agon (cnn_cnn) scores 99.1% of oracular performance (97.6-99.1% for all Agon flavors). In comparison, the best always strategy (always-greedy) scores 95.1%.

In conclusion, our results reveal that a competitive scheduling approach like Agon is more beneficial when used with difficult-to-schedule systems such as H8, and with high-performance systems such as HP8.

4.8 Conclusions

This Chapter presents Agon, a neural network-based competitive scheduler, which successfully selects the most efficient scheduler from a list of six candidates, according to the different instances of the scheduling problem it is faced with. Agons architecture includes a novel denoising autoencoder front-end capable of minimizing the prediction error introduced at its input by the system's performance prediction module. Our evaluation methodology demonstrates Agon to outperform single-scheduler approaches in all our benchmark systems, while achieving performance within 1% of an oracular scheduler selector.

Acknowledgements

Chapter 4, in full, is currently being prepared for submission for publication of the material. Prodromou, Andreas; Venkat, Ashish; Tullsen, Dean M., Agon: A Scalable Competitive Scheduler. The dissertation author was the primary investigator and author of this material.

Chapter 5

Concluding Remarks

Heterogeneous hardware is becoming increasingly more available in modern hardware. Over the years, more and more heterogeneous products appear in the market, while research breakthroughs enforce the expectation that heterogeneity will increase in the future. Today, one can find commercial products with heterogeneous memory architectures, and (single-ISA for the moment) heterogeneous multicore architectures. Recent research proposals further demonstrate that even the Instruction Set Architecture can vary in future generations of these products, dubbed "Heterogeneous-ISA multicore architectures".

Allowing hardware to be heterogeneous, has been shown in the literature to lead to significant gains in performance and power consumption, assuming intelligent scheduling and resource management. However, as the amount of heterogeneity within systems increases, the scheduling problem becomes a significant challenge. Existing proposals lead to poor utilization of a system's structures, incur significant cost due to their high overhead, and do not scale well to higher core counts.

This dissertation demonstrates the necessity and impact of accurate predictions in the presence of heterogeneous hardware and explores algorithms, methodologies and microarchitectural modifications towards this pursuit. System schedulers can utilize a predicted view of the near future and intelligently allocate resources to executing software.

Utilizing a Majority Element Algorithm proposed for big data management, we were able to implement an extremely lightweight and high accuracy predictor for dynamic memory activity patterns, which is demonstrated to significantly outperform typically used, counter-based predictors. This dissertation proposes MemPod as a manager for heterogeneous memory architectures, which couples this novel predictor with our proposed clustering of memory channels into independent Pods. Our evaluation demonstrates MemPod to improve average main memory access time by proactively migrating soon-to-be-accessed pages into fast memory.

This dissertation further presents proposals for machine learning-based performance predictors for heterogeneous-ISA multicore architectures. The proposed predictors are trained to capture the subtle interactions between software and hardware, allowing them to extrapolate measurements from a “reference core”, to runtime performance when the same software executes on any other core configuration from our dataset. We propose three novel metrics that allow us to set solid requirements on the quality our performance predictors should have in order to be useful. We are able to utilize our findings and keep our predictors simpler without a negative impact on system performance. This work unlocks performance prediction that for the first time remains accurate across general purpose ISAs.

With the ability to “see the future” via accurate performance predictions, we shifted our focus on the last piece of the puzzle: the scheduler itself. We propose Agon, our scalable, competitive scheduler. This neural network-based proposal, decides which scheduling algorithm should operate at any given moment, in such a way that it chooses the scheduler with the lowest possible overhead, which should still result in high performance. While it’s true that higher-overhead schedulers tend to result in higher system performance, a combination of the underlying hardware heterogeneity, the set of software waiting to be executed, and the small but unavoidable performance prediction error at the scheduler’s input provide ample opportunity to save on scheduling overhead. Our proposed neural network architecture for Agon also includes a denoising autoencoder front-end which is shown to successfully “repair” predictions and significantly reduce error, very similarly to image denoising approaches.

In conclusion, this dissertation tackles one crucial aspect of execution on heterogeneous, and even heterogeneous-ISA hardware. We improve memory access time, we are able to assign workloads to the appropriate cores such that overall system performance is maximized, and we are able to predict the correct scheduler for a given instance.

Bibliography

[1] The Benefits of Multiple CPU Cores in Mobile Devices. Technical report, NVidia, 2010.

[2] Variable SMP A Multi-Core CPU Architecture for Low Power and High Performance. Technical report, NVidia, 2011.

[3] Qualcomm snapdragon 810 processor: The ultimate connected mobile computing processor. Technical report, Qualcomm, 2014.

[4] Apple 2017: The iphone x (ten) announced. 2017.

[5] Ayaz Akram. A study on the impact of instruction set architectures on processor performance. 2017.

[6] Ayaz Akram and Lina Sawalha. The impact of isas on performance. In *WDD'14*.

[7] AMD. AMD Radeon^{TM}R9 series graphics cards with high-bandwidth memory. http://www.amd.com/en-us/products/graphics/desktop/r9. Accessed: 2016-04-01.

[8] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance. In *MICRO'48*, pages 725–737. ACM, 2015.

[9] Ioana Baldini, Stephen J Fink, and Erik Altman. Predicting gpu performance from cpu runs using machine learning. In *SBAC-PAD*, pages 254–261. IEEE, 2014.

[10] Antonio Barbalace, Rob Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-ren Chuang, and Binoy Ravindran. Breaking the boundaries in heterogeneous-isa datacenters. In *ASPLOS*, 2017.

[11] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the programmability gap in heterogeneous-isa platforms. In *ECCS*, page 29. ACM, 2015.

[12] Jon A Benediktsson, Philip H Swain, and Okan K Ersoy. Neural network approaches versus statistical methods in classification of multisource remote sensing data. 1990.

[13] Sharath K Bhat, Ajithchandra Saya, Hemedra K Rawat, Antonio Barbalace, and Binoy Ravindran. Harnessing energy efficiency of heterogeneous-isa platforms. *ACM SIGOPS Operating Systems Review*, 49(2):65–69, 2016.

[14] Bryan Black. Die Stacking is Happening. In *Proc. of the Intl. Symp. on Microarchitecture*, Davis, CA, December 2013.

[15] Fred A Bower, Daniel J Sorin, and Landon P Cox. The impact of dynamically heterogeneous multicore processors on thread scheduling. *IEEE micro*, 28(3):17–25, 2008.

[16] Björn B Brandenburg, John M Calandrino, and James H Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *2008 Real-Time Systems Symposium*, pages 157–169. IEEE, 2008.

[17] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[18] Stephen D Brown, Robert J Francis, Jonathan Rose, and Zvonko G Vranesic. *Field-programmable gate arrays*, volume 180. Springer Science & Business Media, 2012.

[19] Rainer Buchty, Vincent Heuveline, Wolfgang Karl, and Jan-Philipp Weiss. A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators. *Concurrency and Computation: Practice and Experience*, 24(7):663–675, 2012.

[20] Rainer E Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment problems*. Springer, 2009.

[21] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 52. ACM, 2011.

[22] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

[23] Jian Chen and Lizy K John. Efficient program scheduling for heterogeneous multi-core processors. In *2009 46th ACM/IEEE Design Automation Conference*, pages 927–930. IEEE, 2009.

[24] Jian Chen, Nidhi Nayyar, and Lizy K John. Mapping of applications to heterogeneous multi-cores based on micro-architecture independent characteristics. In *Third Workshop on Unique Chips and Systems, ISPASS2007. April 2007*, 2017.

[25] Dazhao Cheng, Changjun Jiang, and Xiaobo Zhou. Heterogeneity-aware workload placement and migration in distributed sustainable datacenters. In *IPDPS*, 2014.

[26] Nagabhushan Chitlur, Ganapati Srinivasa, Scott Hahn, Pragya K Gupta, Dheeraj Reddy, David Koufaty, Paul Brett, Abirami Prabhakaran, Li Zhao, Nelson Ijih, et al. Quickia: Exploring heterogeneous architectures on real prototypes. In *HPCA*. IEEE, 2012.

[27] Chiachen Chou, Aamer Jaleel, and Moinuddin 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-47, 2014.

[28] Chiachen Chou, Aamer Jaleel, and Moinuddin K Qureshi. Bear: techniques for mitigating bandwidth bloat in gigascale dram caches. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 198–210. ACM, 2015.

[29] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Charm: A composable heterogeneous accelerator-rich microprocessor. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pages 379–384. ACM, 2012.

[30] Digital Equipment Corporation. Alpha architecture reference manual.

[31] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *ISCA*, pages 213–224, June 2012.

[32] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. Execution migration in a heterogeneous-isa chip multiprocessor. In *ASPLOS XVII*, 2012.

[33] Yu Du, Miao Zhou, Bruce R Childers, Daniel Mossé, and Rami Melhem. Supporting superpages in non-contiguous physical memory. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 223–234. IEEE, 2015.

[34] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115, 2017.

[35] Sushmito Ghosh and Douglas L Reilly. Credit card fraud detection with a neural-network. In *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*, volume 3, pages 621–630. IEEE, 1994.

[36] Lovedeep Gondara. Medical image denoising using convolutional denoising autoencoders. In *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, pages 241–246. IEEE, 2016.

[37] Peter Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, pages 1–8, 2011.

[38] Nagendra Gulur, Mahesh Mehendale, R Manikantan, and R Govindarajan. Bi-modal dram cache: Improving hit rate, hit latency and bandwidth. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 38–50. IEEE, 2014.

[39] Akihiro Hayashi, Kazuaki Ishizaki, Gita Koblents, and Vivek Sarkar. Machine-learning-based performance heuristics for runtime cpu/gpu selection. In *PPPJ*, 2015.

[40] Katrin Heitmann, Nicholas Frontiere, Chris Sewell, Salman Habib, Adrian Pope, Hal Finkel, Silvio Rizzi, Joe Insley, and Suman Bhattacharya. The q continuum simulation: harnessing the power of gpu accelerated supercomputers. *The Astrophysical Journal Supplement Series*, 219(2):34, 2015.

[41] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[42] M.D. Hill and M.R. Marty. Amdahl's Law in the Multicore Era. *Computer*, July 2008.

[43] Jonathan Hines. Stepping up to summit. *Computing in Science & Engineering*, 20(2):78–82, 2018.

[44] Arthur E Hoerl and Robert W Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[45] Intel. Intel 64 and ia-32 architectures software developers manual.

[46] Intel. KnightsLanding. http://www.realworldtech.com/knights-landing-details/.

[47] JEDEC. Wide I/O Single Data Rate (Wide I/O SDR). http://www.jedec.org/standards-documents/docs/jesd229.

[48] Djordje Jevdjic, Gabriel H Loh, Cansu Kaynak, and Babak Falsafi. Unison cache: A scalable and effective die-stacked dram cache. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 25–37. IEEE, 2014.

[49] Joint Electron Devices Engineering Council. JEDEC: 3D-ICs. http://www.jedec.org/category/technology-focus-area/3d-ics-0.

[50] Joint Electron Devices Engineering Council. JEDEC: High Bandwidth Memory (HBM) DRAM. https://www.jedec.org/standards-documents/results/jesd235A.

[51] Roy Jonker and Ton Volgenant. Improving the hungarian assignment algorithm. *Operations Research Letters*, 5(4):171–175, 1986.

[52] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760*, 2017.

[53] Jim Kahle. The cell processor architecture. In *MICRO*, page 3. IEEE Computer Society, 2005.

[54] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman Unsal. Redundant memory mappings for fast access to large memories. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 66–78. IEEE, 2015.

[55] Richard M Karp, Scott Shenker, and Christos H Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems (TODS)*, 28(1):51–55, 2003.

[56] Ali Khajeh-Saeed and J Blair Perot. Direct numerical simulation of turbulence using gpu accelerated supercomputers. *Journal of Computational Physics*, 235:241–257, 2013.

[57] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. 2015.

[58] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[59] Sotiris Kotsiantis, Dimitris Kanellopoulos, Panayiotis Pintelas, et al. Handling imbalanced datasets: A review. *GESTS International Transactions on Computer Science and Engineering*, 30(1):25–36, 2006.

[60] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[61] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[62] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO*, 2003.

[63] Rakesh Kumar, Dean M Tullsen, Parthasarathy Ranganathan, Norman P Jouppi, and Keith I Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA*, 2004.

[64] Wooseok Lee, Dam Sunwoo, Christopher D Emmons, Andreas Gerstlauer, and Lizy K John. Exploring heterogeneous-isa core architectures for high-performance and energy-efficient mobile socs. In *GLSVLSI*, 2017.

[65] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W Lee. A fully associative, tagless dram cache. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 211–222. ACM, 2015.

[66] ARM Limited. Arm7/tdmi technical reference manual.

[67] Guangshuo Liu, Jinpyo Park, and Diana Marculescu. Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems. In *2013 IEEE 31st international conference on computer design (ICCD)*, pages 54–61. IEEE, 2013.

[68] Gabriel H. Loh. 3d-stacked memory architectures for multi-core processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 453–464, Washington, DC, USA, 2008. IEEE Computer Society.

[69] Gabriel H Loh and Mark 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*, pages 454–464. ACM, 2011.

[70] Kai Ma, Xue Li, Ming Chen, and Xiaorui Wang. Scalable power control for many-core architectures running multi-threaded applications. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 449–460. ACM, 2011.

[71] Roland Memisevic, Christopher Zach, Marc Pollefeys, and Geoffrey E Hinton. Gated softmax classification. In *Advances in neural information processing systems*, pages 1603–1611, 2010.

[72] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, February 2015.

[73] Micron. NVDIMM. https://www.micron.com/products/dram-modules/nvdimm#/, 2015. Accessed: 2016-04-01.

[74] Sparsh Mittal and Jeffrey S Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):69, 2015.

[75] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. *Introduction to linear regression analysis*, volume 821. John Wiley & Sons, 2012.

[76] Sreerama K Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data mining and knowledge discovery*, 2(4):345–389, 1998.

[77] Prashant J Nair, David A Roberts, and Moinuddin K Qureshi. Citadel: Efficiently protecting stacked memory from large granularity failures. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 51–62. IEEE Computer Society, 2014.

[78] Ripal Nathuji, Canturk Isci, and Eugene Gorbatov. Exploiting platform heterogeneity for power efficient data centers. In *ICAC*, 2007.

[79] John Neter, Michael H Kutner, Christopher J Nachtsheim, and William Wasserman. *Applied linear statistical models*, volume 4. Irwin Chicago, 1996.

[80] John Nickolls and William J Dally. The gpu computing era. *IEEE micro*, 30(2):56–69, 2010.

[81] Joel Nider and Mike Rapoport. Cross-isa container migration. In *SYSTOR*, 2016.

[82] NVIDIA. NVIDIA Pascal. http://devblogs.nvidia.com/parallelforall/nvlink-pascal-stacked-memory-feeding-appetite-big-data/.

[83] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*, 2018.

[84] J. Thomas Pawlowski. Hybrid Memory Cube: Breakthrough DRAM Performance with a Fundamentally Re-Architected DRAM Subsystem. In *Proc. of the 23rd Hot Chips*, Stanford, CA, August 2011.

[85] J Thomas Pawlowski. Hybrid memory cube (hmc). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–24. IEEE, 2011.

[86] PCM. PCM. http://www.extremetech.com/extreme/182096-ibm-demonstrates-next-gen-phase-change-memory-thats-up-to-275-times-faster-than-your-ssd, 2015. Accessed: 2016-04-01.

[87] Andreas Prodromou, Ashish Venkat, and Dean M Tullsen. Deciphering predictive schedulers for heterogeneous-isa multicore architectures. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 51–60. ACM, 2019.

[88] Andreas Prodromou, Ashish Venkat, and Dean M Tullsen. Platform-agnostic learning-based scheduling. In *Proceedings of the 19th International Conference on Embedded-Computer Systems: Architecture MOdeling and Simulation (SAMOS)*, 2019.

[89] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*, 35(3):10–22, 2015.

[90] Moin Qureshi and Gabriel H. Loh. Fundamental Latency Trade-offs in Architecturing DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design. In *Proc. of the 45th Intl. Symp. on Microarchitecture*, Vancouver, Canada, December 2012.

[91] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[92] Payam Refaeilzadeh, Lei Tang, and Huan Liu. Cross-validation. In *Encyclopedia of database systems*, pages 532–538. Springer, 2009.

[93] scikit learn. Scikit-learn python library. http://scikit-learn.org/stable/.

[94] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. Hass: a scheduler for heterogeneous multicore systems. *ACM SIGOPS Operating Systems Review*, 43(2):66–75, 2009.

[95] Takashi Shimokawabe, Takayuki Aoki, Tomohiro Takaki, Toshio Endo, Akinori Yamanaka, Naoya Maruyama, Akira Nukada, and Satoshi Matsuoka. Peta-scale phase-field

simulation for dendritic solidification on the tsubame 2.0 supercomputer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 3. ACM, 2011.

[96] Jaewoong Sim, Alaa R. Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. Transparent hardware management of stacked dram as part of memory. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, 2014.

[97] Michael John Sebastian Smith. *Application-specific integrated circuits*. Addison-Wesley Professional, 2008.

[98] Thannirmalai Somu Muthukaruppan, Anuj Pathania, and Tulika Mitra. Price theory based power management for heterogeneous multi-cores. In *ASPLOS '14*. ACM, 2014.

[99] John E Stone, Kirby L Vandivort, and Klaus Schulten. Gpu-accelerated molecular visualization on petascale supercomputing platforms. In *Proceedings of the 8th International Workshop on Ultrascale Visualization*, page 6. ACM, 2013.

[100] Radu Teodorescu and Josep Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *ACM SIGARCH computer architecture news*, volume 36, pages 363–374. IEEE Computer Society, 2008.

[101] C. Torng, M. Wang, and C. Batten. Asymmetry-aware work-stealing runtimes. In *ISCA 2016*, 2016.

[102] Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, and Onur Mutlu. Dash: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):65, 2016.

[103] Han Vanholder. Efficient inference with tensorrt, 2016.

[104] SMP Variable. A multi-core cpu architecture for low power and high performance. *Whitepaper-http://www. nvidia. com*, 2011.

[105] Ashish Venkat. *Breaking the ISA Barrier in Modern Computing*. PhD thesis, UC San Diego, 2018.

[106] Ashish Venkat, Harsha Basavaraj, and Dean M. Tullsen. Composite-isa cores: Enabling multi-isa heterogeneity using a single isa. In *HPCA 2019*, 2019.

[107] Ashish Venkat, Sriskanda Shamasunder, Hovav Shacham, and Dean M. Tullsen. Hipstr: Heterogeneous-isa program state relocation. In *ASLPOS*, 2016.

[108] Ashish Venkat and Dean M. Tullsen. Harnessing ISA diversity: Design of a heterogeneous-isa chip multiprocessor. In *ISCA*, 2014.

[109] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

[110] Stavros Volos, Javier Picorel, Babak Falsafi, and Boris Grot. Bump: Bulk memory access prediction and streaming. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 545–557. IEEE Computer Society, 2014.

[111] Jonathan A Winter and David H Albonesi. Scheduling algorithms for unpredictably heterogeneous cmp architectures. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 42–51. IEEE, 2008.

[112] Jonathan A Winter, David H Albonesi, and Christine A Shoemaker. Scalable thread scheduling and global power management for heterogeneous many-core architectures. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 29–39. IEEE, 2010.

[113] Gene Wu, Joseph L Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. Gpgpu performance and power estimation using machine learning. In *HPCA*, 2015.

[114] William A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

[115] Xinnian Zheng, Lizy K John, and Andreas Gerstlauer. Accurate phase-level cross-platform power and performance estimation. In *DAC*, 2016.