The Dissertation Committee for Reena Panda
certifies that this is the approved version of the following dissertation:

Accurate Modeling of Core and Memory Locality for Proxy Generation Targeting Emerging Applications and Architectures

Committee:

Lizy K. John, Supervisor

Earl E. Swartzlander Jr.

Sarfraz Khurshid

Andreas Gerstlauer

Karthik Ganesan

Accurate Modeling of Core and Memory Locality for Proxy Generation Targeting Emerging Applications and Architectures

by

Reena Panda

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2017

Dedicated to my husband Prateek,

my parents

and

my sister

Acknowledgments

I would like to thank my advisor, Dr. Lizy Kurian John, for her advice, support and guidance. Dr. John had a great influence on the overall direction of this research and the content of this dissertation. Without her guidance over the years, this work would not have been possible. I am grateful to her for making my research experience a memorable one.

I would also like to thank my committee members, Professor Andreas Gerstlauer for his helpful comments and feedback in improving many research papers that I submitted as a graduate student; Professor Sarfraz Khurshid for his support and encouragement in my most difficult time in graduate school; Professor Earl Swartzlander for his kindness and useful suggestions during the course of this work; and Dr. Karthik Ganesan for providing me valuable feedback to improve this dissertation. Special thanks to Prof. Paul Gratz for introducing me to the field of computer architecture research and motivating me to pursue a Ph.D.

Thanks to Melanie, Melody, Barry and other administrative assistants who worked in the ECE department in the past years.

I also thank my fellow group members at the Laboratory for Computer Architecture (LCA). I enjoyed my collaborations with Dr. Xinnian Zheng, Chris Erb, Michael Lebeane, Jiajun Wang, Wooseok Lee, Dr. Jee Ho Ryoo, Shuang Song and Qinzhe Wu on various research projects. Their research ideas, their knowledge in many diverse disciplines, willingness to listen and comment in many talks and discussions, and friendly help were much appreciated. I also enjoyed great discussions with Mochamad Asri and Kisore Punniyamurthy regarding research and otherwise.

I want to dedicate this dissertation to my father (bapa). Bapa has taught me the value of hard work and perseverance. He is and will always be my first teacher. I would like to thank my mother for her unconditional love, support and encouragement all through my life. She is my biggest strength. I would also like to thank my sister for being there with me through every good and bad phase of life and always motivating me to keep trying. I am eternally grateful to my family for everything. I would also like to thank my in-laws for being very supportive and encouraging throughout my graduate studies.

This dissertation would not be complete without giving credit to my husband and best friend, Prateek Srivastava. His unwavering support, constant motivation and enthusiasm has kept me going through the difficult years of graduate school. He took active interest in my research, helped me refine and shape ideas, proofread my papers, and gave feedback on my presentations. This is not something I could have accomplished alone.

Lastly, I would like to thank god for blessing me with this opportunity and giving me the strength to reach this point.

Accurate Modeling of Core and Memory Locality for Proxy Generation Targeting Emerging Applications and Architectures

Publication No. ___________________

Reena Panda, Ph.D.
The University of Texas at Austin, 2017

Supervisor: Lizy K. John

Designing optimal computer systems for improved performance and energy efficiency requires architects and designers to have a deep understanding of the end-user workloads. However, many end-users (e.g., large corporations, banks, defense organizations, national labs, etc.) are apprehensive to share their applications with designers due to the confidential nature of software code and data. In addition, emerging applications pose significant challenges to early design space exploration due to their long-running nature and the highly complex nature of their software stack that can not be supported on many early performance models.

The above challenges can be overcome by using a proxy benchmark. A miniaturized proxy benchmark can be used as a substitute of the original workload to perform early computer performance evaluation. The process of generating a proxy benchmark consists of extracting a set of key statistics to summarize the behavior of end-user applications through profiling and using the collected statistics to synthesize a representative proxy benchmark. Using such proxy benchmarks can help designers to understand the behavior of end-user's workloads in a reasonable time without the users having to disclose sensitive information about their workloads.

Prior proxy benchmarking schemes leverage micro-architecture independent metrics, derived from detailed simulation tools, to generate proxy benchmarks. However, many emerging workloads do not work reliably with many profiling or simulation tools, in which case it becomes impossible to apply prior proxy generation techniques to generate proxy benchmarks for such complex applications. Furthermore, these techniques model instruction pipeline-level locality in great detail, but abstract out memory locality modeling using simple stride-based models. This results in poor cloning accuracy especially for emerging applications, which have larger memory footprints and complex access patterns. A few detailed cache and memory locality modeling techniques have also been proposed in literature. However, these techniques either model limited locality metrics and suffer from poor cloning accuracy or are fairly accurate, but at the expense of significant metadata overhead. Finally, none of the prior proxy benchmarking techniques model both core and memory locality with high accuracy. As a result, they are not useful for studying system-level performance behavior. Keeping the above key limitations and shortcomings of prior work in mind, this dissertation presents several techniques that expand the frontiers of workload proxy benchmarking, thereby enabling computer designers to gain a better and faster understanding of end-user application behavior without compromising the privileged nature of software or data.

This dissertation first presents a core-level proxy benchmark generation methodology that leverages performance metrics derived from hardware performance counter measurements to create miniature proxy benchmarks targeting emerging big-data applications. The presented performance counter based characterization and associated extrapolation into generic parameters for proxy generation enables faster analysis (runs almost at native hardware speeds, unlike prior workload cloning proposals) and proxy generation for emerging applications that do not work with simulators or profiling tools. The generated proxy benchmarks are representative of the performance of the real-world big-data applications, including operating system and run-time effects, and yet converge to results quickly without needing any complex software stack support.

Next, to improve upon the accuracy and efficiency of prior memory proxy benchmarking techniques, this dissertation presents a novel memory locality modeling technique that leverages localized pattern detection to create miniature memory proxy benchmarks. The presented technique models memory reference locality by decomposing an application's memory accesses into a set of independent streams (localized by using address region based localization property), tracking fine-grained patterns within the localized streams and, finally, chaining or interleaving accesses from different localized memory streams to create an ordered proxy memory access sequence. This dissertation further extends the workload cloning approach to Graphics Processing Units (GPUs) and presents a novel proxy generation methodology to model the inherent memory access locality of GPU applications, while also accounting for the GPU's parallel execution model. The generated memory proxy benchmarks help to enable fast and efficient design space exploration of futuristic memory hierarchies.

Finally, this dissertation presents a novel technique to integrate accurate core and memory locality models to create system-level proxy benchmarks targeting emerging applications. This is a new capability that can facilitate efficient overall system (core, cache and memory subsystem) design-space exploration. This dissertation further presents a novel methodology that exploits the synthetic benchmark generation framework to create hypothetical workloads with performance behavior that does not currently exist. Such proxies can be generated to cover anticipated code trends and can represent futuristic workloads before the workloads even exist.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer system design and research heavily relies on software simulation techniques to measure the performance of design alternatives and evaluate different design trade-offs. A processor simulator typically applies a workload and/or its input data-set to a model of the target processor's architecture and simulates the execution of individual instructions of the workload on the processor model, while recording its effect on various components of the underlying architecture.

Processor simulation tools range from microarchitectural-level performance simulators (e.g., Gem5 [13], Marssx86 [77]) to detailed register-transfer-level (RTL) models. RTL simulators simulate a very precise model of a processor that are sufficiently detailed to be manufactured. Modeling at such detailed abstraction levels ensures higher simulation accuracy, but it comes at the expense of significantly slow simulation speeds. On the other hand, performance simulators are generally higher-level and more versatile, and they can estimate program performance and statistics (e.g., number of branch mispredictions, various types of cache misses, TLB misses) with reasonably good accuracy at higher simulation speeds. Performance simulators can be further categorized into execution-driven simulators that execute a complete program binary (with or without an input data-set) on a performance model and trace-driven simulators that execute a trace containing partial information about individual instructions and addresses. Different types of performance simulation models have been developed for solving different research and design problems, each model with its own simulation speed and accuracy trade-offs. Such models play a key role in performing design space exploration, evaluating new architectural ideas and identifying the performance bottlenecks of various designs by enabling architects to simulate representative target workloads.

The choice of workloads to simulate on a processor model depends on the class of applications that the architecture targets or the different features of the architecture that the designer wants to stress using the applications. Identifying the right set of workloads to simulate on a performance model is a very challenging problem. Small hand-written micro-kernels can be used to represent commonly-used algorithms in real-world applications, but such micro-benchmarks are often not comprehensive enough to be representative of the real target applications. A set of standard benchmarks are typically used for performing computer design space exploration. For example, Standard Performance Evaluation Corporation (SPEC) creates the popular SPEC CPU benchmarks (e.g., SPEC CPU2000 [92], SPEC CPU2006 [93], SPEC CPU2017 [94]), which represent the most commonly used general-purpose CPU applications and are widely used by computer architecture researchers and designers. The growth in complexity and popularity of data-management applications, fueled by the big-data revolution, has led to creation of a number of big-data benchmark suites such as the Cloudsuite [26], BigDataBench [31], Yahoo! Cloud Serving (YCSB) [18] and TPC-H [100] benchmarks. More recently, graphics processing units (GPUs) have emerged as a popular computation platform for applications beyond graphics. Programmers exploit these massively parallel architectures in diverse domains (e.g., linear algebra and bio-informatics). This has led to several general-purpose GPU benchmark suite offerings such as Rodinia [17], Parboil [96], etc. Given the growing diversity and complexity of the applications that are run on modern-day systems, it is important to evaluate the architectural implications and performance/power efficiency of future computer designs when targeting such emerging applications and architectures.

Unfortunately, detailed performance evaluation and benchmarking of computer systems is a very challenging task. The growing complexity of modern workloads and computer systems pose even bigger challenges. For example, the slow speed of simulation models, the long-running nature of workloads, the complexity of software stacks of emerging workloads and confidentiality concerns regarding sharing end-user applications/data significantly limit the efficiency of early design space exploration studies. This dissertation focuses on developing techniques that can address the above challenges and help computer designers to have a better understanding of end-user workloads. The techniques rely on proxy benchmarking, i.e., replicating the performance behavior of end-user applications using miniaturized proxy benchmarks. The proxy benchmarks do not have complex software stack dependencies and can be used for computer design space exploration without compromising the privileged nature of software, while significantly reducing the simulation times. The presented proxy benchmarking techniques can improve the early design space exploration efficiency of emerging applications and architectures.

1.1 Problem Description

Data handling and management has become an integral component of all businesses, big or small. Every major industrial sector, be it health-care, scientific-computing, retail, telecommunication or social networking generates large amounts of data every day. This has led to an unprecedented increase in the demand for efficient data handling systems and applications. While traditional data management systems were based on structured-query language (SQL) based relational databases, a new group of databases popularly known as NoSQL databases have recently emerged as competitive alternatives owing to their flexibility and scalability properties [37, 27, 18, 14]. Recent years have thus, seen a big surge in several such emerging application domains e.g., database applications, machine learning, business analytics etc. Big-data processing needs also challenge the capabilities of traditional computing systems to process the large amounts of data efficiently in terms of both performance and power. Thus, computer designers need to re-evaluate their design principles to target such emerging applications [26, 31, 67, 70]. However, it is very challenging to study the performance and power behavior of complex emerging applications by running them on early performance models. The following paragraphs summarize the key challenges and considerations affecting early design-space exploration efficiency.

The first challenge is that it is very difficult (and often impossible) to run many emerging applications on detailed performance models owing to the complex application software stacks and significantly long run times of such applications. Figure 1.1 shows the software stack of a typical web-serving engine, consisting of

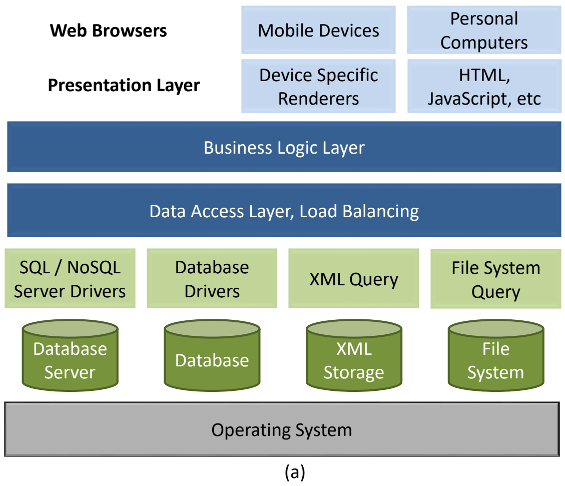


Figure 1.1: Software Stack of a Typical Web-serving Engine

layers of complicated software levels interacting together to form the backbone of the engine. Running similar applications requires handling the complex software layers, back-end databases, third-party libraries, etc., which are quite challenging to support on most early performance models. Typically, a set of standard benchmarks [93, 92, 95, 100, 99] are used for performing computer performance evaluation. Benchmarks such as SPEC CPU2006 [93] and Linpack [36] are comparatively simpler targets for performance evaluation and are widely used by the computer architecture research community. Figure 1.2 compares the performance correlation between three emerging database applications (YCSB benchmarks running with Cassandra [16] and MongoDB [56] databases and TPC-H benchmarks running with MySQL [58] database) with three widely popular benchmark suites (SPEC CPU2006 [93], SPECjbb2013 [95] and Linpack [36]) across six key performance metrics: L1 data cache (L1D) misses per kilo instructions (MPKI), L1 instruction cache MPKI (L1I), last-level cache (LLC) MPKI and translation look-aside buffer

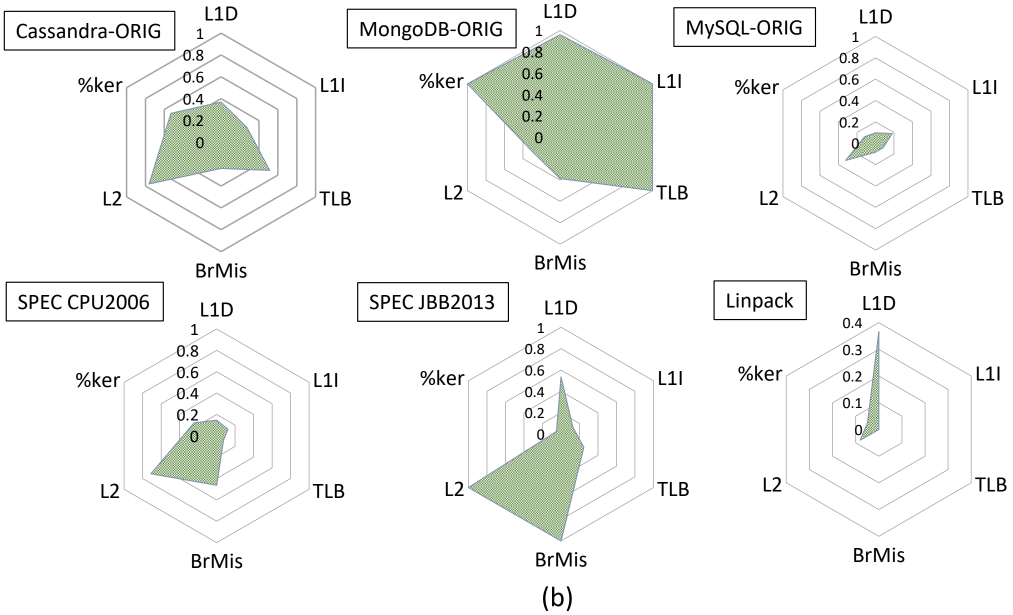


Figure 1.2: Kiviat Plots Comparing Performance of Emerging Big-data Applications with Standard Benchmarks

(TLB) MPKI, fraction of kernel instructions executed (%Ker) and branch misprediction rate (BrMis). The plots clearly illustrate that significant diversity exists in the performance and bottlenecks of the emerging applications and standard benchmarks [70, 67, 104]. On the other hand, the recently proposed big-data benchmarks (e.g., Cloudsuite and BigDataBench) suffer from a problem similar to those of the real-world applications; they rely on the ability of early performance models to support complex software stacks with back-end databases.

The next challenge affecting design space exploration is that simulation speeds are orders of magnitude slower than native execution. This severely limits the efficiency of extensive design space exploration studies. To complicate matters further, most emerging applications have significantly long run times. For example, the average dynamic instruction counts of the YCSB benchmarks (with back-end Cassandra database) and the SPEC CPU2017 benchmarks are in the range of tens of trillions of instructions. Simulating even one such benchmark for a single design point can easily take anywhere between several days to several weeks to finish. Evaluating the entire design search space of a single architecture, on multiple benchmarks, will require up to thousands of simulation hours. That is equivalent to several months or even years of simulation on a single processor. Simulating large-scale processors running such complex applications is well beyond the capabilities of even the fastest and fanciest simulators, even on today's fast machines.

Finally, it is difficult to get access to many end-user applications due to confidentiality concerns regarding sharing software code or data. For example, such software may include weapons simulation, proprietary market trading algorithms, trade secrets or other sensitive data. Thus, on one hand, designers need in-depth insights into end-user workload behavior, and on the other hand, end users cannot reveal any significant information about their code to the designers due to the proprietary or confidential nature of their software/data.

1.1.1 Limitations of Prior Research Work

A lot of research has been done to address the above challenges affecting efficient design space exploration of future computer designs. Benchmark sampling techniques such as simulation points [87] and SMARTS [108] have been proposed to reduce simulation time requirements of a benchmark. Such techniques analyze phase-level redundancies within a single workload's execution and identify workload sub-sections to represent the overall execution behavior of the entire workload. By doing so, these techniques can address the long simulation time problem. However, they still suffer from the challenge of supporting complex application software-stacks on early performance models and they do not address the issue of proprietary end-user applications. Also, they rely on the capability of simulators to fast-forward execution until the interval of interest. Other techniques such as benchmark subsetting [79] have been proposed to identify a subset of benchmarks from a benchmark suite that can represent the performance behavior of all the benchmarks within the suite. However, the subset results are still whole programs and are still too big, complex and proprietary to be directly used with performance models.

A promising alternative to address the above challenges is proxy benchmarking. Proxy benchmarking is the process of extracting statistics that summarize the behavior of the end-user workloads through profiling and then using the statistics to synthesize a miniaturized representation of the proprietary workloads (called a **"proxy"** or **"clone"**). The proxy workloads can be used as miniaturized substitutes of the original workloads to perform early computer system performance evaluation, helping designers to understand the behavior of users' workloads in a reasonable time and without the users having to disclose sensitive information about the original workload.

Prior proxy generation proposals [42, 29, 43] utilize an extensive set of micro-architecture independent metrics that are derived from detailed functional simulators or profilers to synthesize the proxy benchmarks. Unfortunately, such metrics are often very difficult to extract for many emerging applications that can not be supported on early simulation models or profilers. For example, several big-data workloads (e.g., Java-based Cassandra) do not work reliably with many detailed simulation models (e.g., gem5) or profiling tools (e.g., Pin [54], Valgrind). In such cases, it becomes impossible to apply prior proxy benchmark generation techniques to generate proxy benchmarks for such emerging applications.

Furthermore, prior proxy generation techniques capture core-level locality metrics in detail, but abstract out memory locality modeling using very simple dominant stride-based models. As a result, these schemes suffer poor cloning accuracy in replicating performance behavior of applications with complex memory access patterns. For example, Figure 1.3 shows the L1 miss rate cloning error of proxies generated using the single dominant stride (SDS) approach (the most commonly used statistic in literature for modeling memory locality patterns in proxies), for a set of data-serving (YCSB) and data-analytics (TPC-H) benchmarks run against a back-end MySQL database. Figure 1.3's x-axis corresponds to different L1 cache

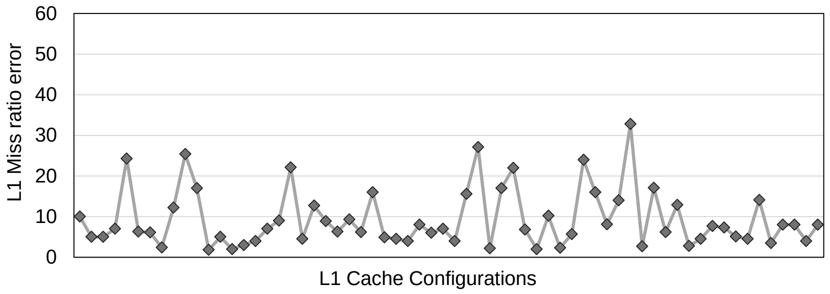


Figure 1.3: L1 Miss-ratio Cloning Error of SDS Proxies for Various L1 Cache Configurations.

configurations, where the cache size and associativity is varied between 16-64KB and 2-8 ways, respectively. It can be observed that the SDS proxies show significant errors in replicating the miss ratio versus the original applications (measured as the absolute difference in miss ratios) at several data points, reaching as high as 33% cloning error. This shows that the SDS approach is not suitable for modeling complex memory access patterns of big-data applications. Most big-data applications are highly data-intensive and their overall system-level performance is significantly impacted by the performance of the cache and memory hierarchy [26, 67]. As a result, prior performance cloning techniques are not very accurate at studying the overall performance of emerging big-data applications.

Several detailed cache and memory cloning techniques [7, 5] have also been proposed in the literature. WEST [7] models temporal locality of applications using per cache-set LRU stack distance distributions based on a baseline cache hierarchy. However, WEST does not model spatial locality behavior, making it inadequate for evaluating microarchitectural structures that exploit spatial locality e.g., prefetchers or the memory system (see Table 1.1 for the cloning error of WEST proxies for over 7000 different prefetcher-enabled last-level cache and TLB configurations across 39 big-data and SPEC CPU benchmarks). Spatio-temporal memory (STM) [5] cloning technique overcomes this limitation and models spatial locality by cap-

Table 1.1: Error between WEST Proxies and Original Applications in terms of Cache and TLB Miss-rates.

	LLC miss rates	TLB miss rates
Average Error	19% (avg)	9.3% (avg)
Maximum Error	44% (max)	22% (max)

turing global stride-based correlations in the memory reference stream. However, the global stride transitions of many benchmarks cannot be captured even by using a stride history depth as long as ∼80-100 [5]. Thus, STM has to maintain long histories in order to capture the dominant global stride transitions, which results in significantly higher meta-data storage overhead. Limiting the stride history depth can reduce storage overhead, but at the expense of significantly poor cloning accuracy. For example, limiting the history length to 40 causes STM proxies to experience up to 24% and 32% error in replicating the TLB miss rate and memory footprint of many big-data applications. Thus, the state-of-the-art memory behavior cloning proposals either model limited locality metrics (e.g., WEST) and suffer from poor cloning accuracy or are fairly accurate, but at the expense of significant meta-data overhead. Additionally, none of these techniques are suitable for modeling the memory access behavior of emerging architectures such as GPUs.

Finally, the above prior proxy generation techniques either model the instruction pipeline-level locality or the memory access locality in detail. However, none of them jointly model both core and memory reference behavior with high accuracy. As a result, they are not suitable for studying system-level performance of emerging applications. This dissertation explores techniques and methods that expand the frontiers of workload proxy benchmark synthesis through more accurate and efficient modeling of core and memory locality behavior of emerging applications and architectures. The proposed techniques can help computer designers to have a better and faster understanding of end-user workload behavior without compromising the privileged nature of software or data.

1.2 Overview of Proposed Research

1.2.1 Core-level Proxy Generation using Performance Counter Based Characterization

While prior system-level proxy benchmarking proposals [11, 43, 29] utilize micro-architecture independent metrics derived from detailed functional simulators for proxy generation, such metrics are often very difficult to generate for many emerging applications. However, it is possible to characterize these workloads based on performance counters. To enable proxy benchmark synthesis for complex, emerging applications that do not work with many simulation models or profiling tools, this dissertation first presents a core-level proxy benchmark synthesis methodology, PerfProx [69, 71], which leverages performance metrics derived from hardware performance counter measurements to create miniature system-level proxy benchmarks. First, the key drivers of big-data application performance (larger code footprints, operating system effects, and other run-time effects, etc.) are identified. Such effects are often not highly significant in traditional desktop or general-purpose applications and thus, are not modeled by prior workload cloning proposals. Then, PerfProx captures the identified performance metrics using hardware performance counters and stochastically models them to create miniaturized proxy benchmarks. The proxy benchmarks replicate the performance behavior of real-world cloud applications, including operating system and run time effects, and yet converge quickly without needing any complex software stack support. Several big-data workloads do not work reliably with many profiling tools, thus performance-counter based characterization and associated extrapolation into generic parameters for the code generator enables fast (runs almost at native hardware speeds, unlike prior workload cloning proposals) and efficient proxy generation for such benchmarks with reasonable accuracy.

1.2.2 Memory Locality Modeling using Hierarchical Pattern Detection

Prior state-of-the-art memory locality modeling proposals [5] create memory proxies by extracting patterns within the global stream of memory references. However, the global memory access streams of applications are shaped by a number of different factors - data-dependent control-flow, high-level algorithms that access different data-structures in the program, data-structure access interleaving, memory layout of data-structures determined by the compiler/runtime and the machine's execution model (out-of-order versus in-order execution, memory address reordering, etc.). As a result, distilling the inherent patterns in the global memory access streams into a small set of statistics is very challenging.

This dissertation argues that memory access locality can be more accurately and succinctly captured by learning patterns at a localized granularity for many applications. With localization, memory addresses are ideally grouped according to some inherent property of programs and data accesses (e.g., code-based, address-region based, time-based), which make the resulting localized streams more predictable. For example, code-based localization exploits the fact that different memory instructions or PCs perform different functions versus other memory instructions, which get executed around the same time. This dissertation presents a novel class of memory proxy generation techniques [74] that exploit pattern detection and modeling at localized granularity to accurately replicate application cache and memory behavior of different applications.

The first memory access locality modeling technique (HALO) exploits address-region localization based pattern tracking to create miniature memory access proxies. HALO discovers patterns by first decomposing an application's memory accesses into a set of independent streams that are constrained to a smaller region of memory and then capturing fine-grained access patterns within localized regions using repeating stride transitions. This allows the representation of complex workloads through the composition of a set of smaller and simpler building blocks. Additionally, different programs have different locality behavior. HALO exploits this observation to achieve higher meta-data storage efficiency by capturing multi-level stride transitions, which are tailored to an application's locality patterns. However, modeling locality within individual streams alone is not sufficient to recreate the original application's memory behavior. There must be a mechanism to combine accesses from these decomposed streams to synthesize an ordered proxy sequence. HALO models this by tracking how accesses to the localized streams are interleaved with respect to each other by using coarse-grained temporal locality tracking. By accurately modeling the spatial locality, temporal locality and memory footprint of application, HALO proxies can replicate the cache and memory performance of applications even with complex memory access patterns.

However, this approach is not sufficient to model the cache and memory access behavior of emerging architectures such as GPUs. GPUs leverage large amounts of parallel hardware combined with light-weight context switching among thousands of threads to hide the impact of long memory latencies. To model the memory locality of GPU applications, it is important to model the effects of thread-level parallelism on the cache and memory hierarchy. This dissertation extends workload cloning techniques to GPUs and presents G-MAP [76], a novel methodology and framework to statistically model the inherent memory access locality and parallelism of GPU applications to create miniaturized GPU memory proxies. G-MAP exploits code-based correlations to model cache and memory locality behavior of GPU applications, while also accounting for GPU's parallel execution model.

1.2.3 System-level Proxy Benchmark Synthesis

Finally, the above techniques (and the prior work in proxy generation) can accurately model either the compute-instruction behavior or the memory access behavior. However, none of the prior cloning studies accurately model the joint performance of both core and memory subsystems and their complex interactions. In reality, the processor core configuration and the application together determine processor core performance, which in turn affects the timing of requests received in the memory system. At the same time, memory performance has a feedback loop with processor performance, which in turn affects the timing of other memory requests and, the overall performance of the application. As a result, the prior workload cloning proposals can not be used for studying overall system-level application performance. However, it would be useful to have easy-to-use and representative benchmarks to study overall system-level performance of emerging applications.

This dissertation presents a novel system-level proxy generation and modeling methodology (CAMP) that accurately models both core performance and memory locality to create miniature proxy benchmarks [73]. To model the processor core performance, the proposed technique captures and models dependencies between instructions (instruction-level parallelism), instruction types, control-flow behavior, etc. An improved memory locality profiling approach is added that accurately captures both the spatial and temporal locality of applications. However, most big-data applications typically do not have a single dominant stride/offset based access pattern. Thus, it is quite difficult to control the different dynamic execution instances of the low-level, static load/store instructions in the proxy benchmark to reproduce the complex memory access patterns of the original applications using synthetic data-structure accesses in the proxy code. To overcome this challenge, this dissertation introduces a novel proxy modeling and replay methodology that integrates the core and memory locality models to create accurate system-level proxy benchmarks.

1.2.4 Synthetic Workloads to Cover Workload Performance Spectrum

Traditionally machines for tomorrow are built using benchmarks of today, which are workloads of yesterday. It is desirable to have benchmarks that model futuristic workloads so that future systems can be designed and tuned to work well for such workloads. This dissertation proposes a synthetic benchmark generation methodology, Genesys that systematically tweaks the program characteristics, used as an input to the proxy generation framework, to produce new hypothetical workloads with performance behavior that does not currently exist. Also, the set of programs included in a standard benchmark suite is limited and the benchmarks often fill only certain data-points in the workload spectrum (most of the spectrum is not represented). Genesys proxies can be generated to cover anticipated code trends and can represent futuristic workloads before the workloads even exist.

1.3 Thesis Statement

Hierarchically capturing both spatial and temporal locality in the application memory streams using inter-region and intra-region access patterns improves the accuracy of modeling memory access behavior of complex emerging applications. Accurately capturing memory access locality and modeling other important features (e.g., system activity) creates more accurate and representative proxy benchmarks. Execution-related metrics can be used for generating proxies of emerging applications, which do not work with conventional profiling tools.

1.4 Thesis Contributions

This dissertation makes several contributions to accurate memory locality and core performance modeling of emerging applications and architectures. The key contributions of this dissertation are summarized as follows.

- To simplify benchmarking of big-data data-serving and data-analytics workloads on early performance models, this dissertation propose to generate miniature, representative proxy benchmarks that do not need any complex software-stack or back-end database support.
- This dissertation presents a core-level proxy benchmark generation method-

ology, which enables fast and efficient proxy generation for emerging big-data applications using performance metrics derived primarily from hardware performance counters. The proxy benchmarks are representative of the performance of the emerging, real-world applications and yet converge to results quickly and do not need any complex software stack support. The presented approach is evaluated using three modern, real-world SQL and NoSQL databases (Cassandra, MongoDB and MySQL) running the data-serving and data-analytics applications on different hardware platforms and with different cache/TLB configurations. The proxy benchmarks closely mimic the performance of the original database applications, while significantly reducing the instruction counts.

- To enable fast and efficient design space exploration of futuristic memory hierarchies, this dissertation next proposes a hierarchical memory access locality modeling technique that identifies patterns in the original memory reference stream by isolating the global memory references into several localized streams and further zooming into each local stream capturing multi-granularity spatial locality patterns. The interleaving degree between localized stream accesses is modeled by leveraging coarse-grained reuse locality patterns. The presented technique is evaluated using over 20,000 different memory system configurations and it achieves over 98.3%, 95.6%, 99.3% and 96% accuracy in performance behavior of replicating prefetcher-enabled L1 & L2 caches, TLB and DRAM performance, respectively. It also outperforms the state-of-the-art memory cloning schemes, WEST and STM, while using $\sim$39X less meta-data storage than STM.

- To enable efficient GPU memory system exploration, this dissertation presents a novel methodology and framework that statistically models the GPU memory access stream locality by exploiting the synergy in code-localized access patterns (within and across threads) to create miniature memory access proxies. To account for the parallel execution model of GPUs, a fine-grained, coordinated scheduling policy is also adopted to ensure appropriate parallelism at the thread-level and cache/memory-level. Extensive evaluation using 18 benchmarks from Rodinia [17], CUDA SDK [62] and Ispass09 [6] benchmark suites shows that the presented technique can mimic the performance of the original GPU workloads with over 90% accuracy across over 5000 L1-cache, L2-cache, prefetcher and DRAM memory configurations.

- The next proposal focuses on synthesizing accurate and representative system-level proxy benchmarks for emerging applications, by modeling both core-performance and memory locality accurately along with modeling the feedback loop between the core and memory performance. Core performance is modeled by capturing metrics such as instruction-level parallelism, control-flow behavior, etc. An improved memory locality profiling approach is added that captures both the spatial and temporal locality of applications. Finally, a novel proxy generation and replay methodology is introduced that integrates the core and memory locality models together to create accurate system-level proxy benchmarks. Using extensive evaluation on a set of big-data database applications, it is demonstrated that the proxies can mimic the original application's overall performance behavior fairly accurately with an average cloning error of 11%. This is a new capability that can enable accurate overall system (core and memory subsystem) design exploration.

- Finally, this dissertation proposes a technique to exploit the synthetic workload generator framework to produce hypothetical workloads before the workloads even exist in order to densely cover the workload performance spectrum. It achieves this by systematically tweaking statistics used as an input to the synthetic workload generator in a systematic manner to produce new hypothetical workloads with performance behavior that does not currently exist.

1.5 Thesis Organization

This dissertation is organized as follows. Chapter 2 provides background about prior proxy benchmark generation and other simulation time reduction techniques. Chapter 3 presents the evaluation framework used in this dissertation and explains the set of benchmarks that were used. Chapter 4 presents details of the proxy generation framework that exploits performance-counter based characterization to generate miniature proxy benchmarks for big-data applications. Chapter 5 presents the hierarchical memory locality modeling approach for generating miniature proxy benchmarks targeting CPU applications. Chapter 6 extends the hierarchical memory locality modeling approach for GPU architectures while adding accountability of GPU's parallel execution model. Chapter 7 presents a unique methodology to combine an accurate memory locality modeling framework with core behavior modeling methodology to create proxy benchmarks which model both core & memory behavior as well as the feedback loop between the two. Chapter 8 presents a methodology to create hypothetical benchmarks by systematically tweaking the workload statistics used as inputs to the proxy generation framework. Chapter 9 concludes this dissertation with a summary of the contributions of the dissertation and suggestions for future research opportunities.

Chapter 2

Related Work

This chapter provides an overview of the state-of-the-art research underlying this dissertation. Broadly, the prior workload cloning proposals can be categorized into techniques that capture detailed cache and memory level behavior of applications and techniques that model detailed core-level performance behavior. The following sections first discuss the respective proxy benchmarking proposals and then highlight a few other simulation time reduction techniques.

2.1 Schemes for Modeling Cache and Memory Performance

The memory reference stream of an application is affected by several factors: high-level algorithms that access different data-structures in the program, memory layout of data-structures determined by the compiler or runtime, program's unique control-flow, machine's execution model (out-of-order versus in-order execution, memory address reordering). As a result, distilling the inherent locality patterns in the memory access streams into a small set of statistics is a very challenging problem. A common approach to capture memory behavior is to start from a model of reference locality. The principle of locality asserts that, whenever a memory address is referenced, the address itself (temporal locality) or addresses near it (spatial locality) are likely to be referenced again soon [90]. Most prior workload cloning approaches exploit some form of these two kinds of locality to model memory access patterns.

Reuse distance is an effective model of temporal locality [55, 64, 20, 7, 107]. It is calculated as the number of unique elements accessed between successive accesses to the same element. Figure 2.1a shows an example of reuse distance computation for the sequence of accesses shown in the figure at cacheline granularity (∞ represents access to a new cacheline). The captured reuse distance distribution can be used to synthesize a trace clone (e.g., {7, 12, 1, 7, 12, 1, 32, 0, 100, 32, 0, 100 ...}), which has the same reuse behavior. However, the synthesized trace sequence does not model the spatial locality behavior of the original sequence and cannot be used for evaluating prefetchers, DRAM, memory footprint. Another approach to model memory access locality is to capture spatial/temporal locality patterns using address or stride transition graphs. An address transition graph records every unique memory address as a node, and each edge connects an address to its successors (see Figure 2.1b). We can observe that while some cache-blocks have a single follower (e.g., 0), others have multiple followers (e.g., 2). The transitions recorded in the graph can be followed to generate a proxy trace. However, the space requirement for saving this graph is often prohibitive. Using stride transition graphs allows us to capture similar patterns in a more compact form (see Figure 2.1c). Nonetheless, longer stride history length correlations need to be exploited for achieving higher accuracy, which makes the storage requirements for capturing global stride transitions also significant.

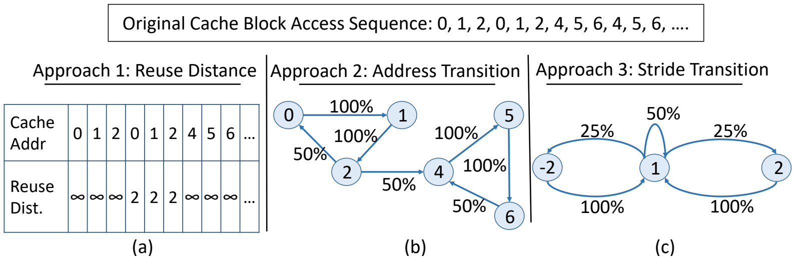

Original Cache Block Access Sequence: 0, 1, 2, 0, 1, 2, 4, 5, 6, 4, 5, 6, ….
Approach 1: Reuse Distance
Cache Addr
0 1 2 0 1 2 4 5 6 …
Reuse Dist.
∞ ∞ ∞ 2 2 2 ∞ ∞ ∞ …
(a)
Approach 2: Address Transition
0
1
2
4
5
6
100%
100%
50%
50%
100%
100%
50%
(b)
Approach 3: Stride Transition
-2
1
2
25%
50%
25%
100%
100%
(c)

Figure 2.1: Common Memory Access Locality Modeling Approaches

WEST [7] is a state-of-the-art data cache performance cloning framework. Based on a baseline L1/L2 cache hierarchy configuration, WEST captures temporal locality patterns using per cache-set LRU stack distance distribution for every level of the profiled cache hierarchy. Each cache-set's stack distance distribution captures the percentage of memory references to the corresponding LRU stack position. WEST also tracks other statistics such as access distribution across different sets, per-set read-write distribution to every stack position. To generate a memory proxy, WEST stochastically samples the stack distance statistics and generates accesses to the chosen cache sets and ways one-by-one. However, WEST's statistics are tightly tied to the profiled cache configuration. Thus, significant deviation between the profiled and test configurations (e.g., cacheline size) leads to poor cloning accuracy. Furthermore, WEST does not model spatial locality patterns, and thus WEST proxies are inadequate to study effects of microarchitectural structures that exploit spatial locality (e.g., prefetchers [68, 45, 91]). Finally, the meta-data overhead of capturing WEST's statistics is proportional to the size of profiled cache hierarchy. Although it is manageable for L1/L2 caches (relatively smaller size), the overhead is high for typically-sized last-level or DRAM caches.

Spatio-Temporal Memory (STM) [5] is another state-of-the-art workload cloning proposal that captures an application's spatial and temporal locality behavior to create memory proxies. STM tracks temporal locality using per cache-set LRU stack distance distribution of a baseline L1 cache (like WEST). For the references that miss in the profiled L1 cache, STM captures their spatial locality patterns by learning global stride transitions (strides following a history of past M strides) in a global stride history table. Since STM captures stride transitions over the global memory sequence, it has to maintain long stride histories to accurately capture dominant patterns. Past research has shown that a history length of as long as ~ 100 is insufficient to capture the access locality of many SPEC CPU2006 benchmarks (e.g., h264ref, wrf, etc.) [5]. Maintaining long history based stride tables significantly increases STM's meta-data storage overhead, which is a key limitation of STM in terms of portability. Limiting the history length can reduce the meta-data overhead, but it increases aliasing in the stride history table resulting in poor cloning accuracy. Furthermore, STM's reliance on a per-set LRU stack distance profile for capturing tight temporal locality patterns causes performance inaccuracies when the target L1 configurations differ from the baseline assumption.

Bell et al. [11], Joshi et al. [43] and Ganesan et al. [29] create workload clones by modeling instruction-level behavior, while modeling memory patterns using a single dominant stride for every load and store instruction. Because of this simplified assumption, these approaches suffer from poor cloning accuracy, especially when modeling complex access patterns of emerging big-data workloads. MEMST [8] clones DRAM performance by modeling statistics such as bank conflict behavior, row buffer hit ratio, etc. and is tightly tied to the profiled DRAM parameters. Metoo [106] generates workload clones by replicating memory traffic timing behavior, but the memory addresses are based on WEST's methodology.

While the above approaches model performance of some elements of the memory hierarchy, the proposed cloning techniques are the first to capture enough spatial and temporal features to model the performance of multi-level caches, prefetchers, TLB and main memory. Also, the state-of-the-art techniques can model memory access locality of CPU applications, but no such solutions exist for cloning GPU memory access patterns. This dissertation also proposes a novel framework to model memory access locality of GPU applications.

2.2 Schemes for Modeling Core-level Performance

Oskin et al. [65] and Eeckhout et al. [22] introduced the idea of statistical simulation. The approach used in statistical simulation is to generate a short synthetic trace from a statistical profile of workload attributes such as basic-block distribution, branch misprediction rate, data/instruction cache miss rates, instruction mix, dependency distance and then simulate the synthetic trace using a statistical simulator. Eeckhout et al. [23] improved statistical simulation by profiling workload attributes at a basic block granularity using statistical flow graphs. Bell et al. [11] improved upon the statistical framework proposed by Eeckhout et al. [22] by profiling applications at runtime and extracting several execution-related metrics to automatically create proxy workloads. Joshi et al. [43, 42] and Ganesan et al. [29] cloned proprietary applications into synthetic proxies for single core systems by extracting micro-architecture independent attributes only. Ganesan et al. [30] added support for generating proxy workloads for multi-threaded applications.

Prior synthetic generator proposals leverage detailed micro-architectural simulators to perform detailed workload characterization. Also, most prior workload cloning studies have focused on general-purpose applications like SPEC CPU2000 [92], SPEC CPU2006 [93] or embedded benchmarks like Implantbench [40]. However, many emerging applications cannot be reliably run and profiled on detailed micro-architectural simulators till completion. Techniques presented in this dissertation enable fast and accurate proxy generation for such complex, emerging workloads by monitoring their complete execution characteristics (with complex software stacks) on real systems using hardware performance counters.

2.3 Techniques for Modeling GPU Workload Performance

Early design space exploration of GPU architectures is traditionally done using detailed, cycle-accurate simulators [6, 80]. Although accurate, simulator speeds are often very slow, which limits efficiency of extensive design space exploration. A few researchers have also proposed analytical models to estimate GPU cache performance. To model L1 cache miss rate, Tang et al. [97] applied reuse distance theory on a single thread-block on a single core by arguing that there is limited reuse across different thread-blocks. Nugteren et al. [61] proposed another GPU L1 cache model. They collected per-warp memory traces and emulated inter-warp parallelism using round-robin scheduling policy before applying an extended reuse distance model (considering cache latencies, MSHRs, etc.). Although such models are fast, their scope is limited to L1 cache performance modeling. In contrast, this dissertation presents a novel performance cloning framework (G-MAP) that can allow extensive exploration of different levels of the GPU memory hierarchy. Other GPU analytical modeling proposals [35, 89] focus on core performance, while using simple abstractions to model memory performance.

Yu et al. [109] proposed a GPU application cloning technique by replicating the instruction mix, control-flow, divergence behavior. Deniz et al. [19] proposed another GPU benchmark synthesis framework by replicating GPU application features such as the instruction throughput, compute resource utilization. Both these studies focus primarily on mimicking instruction-level characteristics, while they capture memory access patterns using abstract and simple models.

2.4 Other Techniques for Reducing Simulation Time

To address the simulation time problem, well-known sampling techniques like simulation points [87] and SMARTS [108] are widely used. Such techniques leverage the observation a program's dynamic behavior is composed of repeated occurrence of several shorter-duration phases. These techniques try to identify such unique fine-grained phases (called simulation points or simpoints), which can be used to represent the entire execution behavior of the program at a considerably reduced simulation time. To identify the dominant phases, the Simpoint tool [87, 33] divides up the entire execution of an application into fixed-length intervals. It uses a signature to represent the activity during each execution interval called a "basic block vector" [86], which is nothing but a sequence of basic block addresses executed during that interval. Then, the K-means clustering algorithm is used to compare the basic-block vectors of different execution intervals and identify dominant clusters. For each cluster, a representative simulation point is chosen that has the minimum distance from the centroid of the cluster. The simulation point is also given a weight according to the number of execution-intervals grouped into the corresponding cluster. The weights of different simulation points are normalized such that they sum up to unity. However, using such techniques for big-data applications requires supporting complete application software stacks on simulation frameworks and fast-forwarding support.

Other techniques such as benchmark subsetting [79] have been proposed to identify subsets of programs belonging to a benchmark suite, that are representative of the overall performance of the entire suite. However, such techniques still face the challenge that the results are still whole programs, which are very complex and long-running to be simulated on early performance simulators.

Chapter 3

Methodology

To evaluate the effectiveness of the proxy benchmarking schemes proposed in this dissertation, a combination of techniques involving measurements on real hardware systems and simulations is used. The simulators used in this dissertation include MacSim [47], an x86 CPU-system simulator, CMP$im [38], a cache-hierarchy simulator and Ramulator [48], a detailed memory system simulator. GPU performance evaluation is performed using GPGPU-Sim [6], a widely popular GPGPU performance simulator. Measurements of different performance and power characteristics on real hardware systems are performed by reading performance or power counters using the Linux Perf [53] and RAPL [41] tools. Characterization and profiling of different CPU and GPU applications for extracting the workload-specific statistics, which are used to feed the proxy generators, is done primarily using heavily-modified Pin instrumentation tools [54] and CUDA-sim simulation infrastructure [6]. In terms of workloads, this dissertation uses a variety of real-world big-data applications (e.g., SQL and NoSQL based data-serving applications represented by the Yahoo Cloud Serving Benchmarks [18], data-analytics applications represented by the TPC-H benchmarks [100], graph analytics benchmarks [26]) and general-purpose benchmarks (such as SPEC CPU2006 [93] and SPEC CPU2017 [94]). For the GPU proxy benchmarking techniques, popular GPU benchmarks from the Rodinia [17], NVIDIA SDK [60] and GPGPU-sim ISPASS-2009 [6] benchmark suites are evaluated. The remainder of this chapter presents an overview of each tool and also a description of the different workloads/benchmark suites used to evaluate the proposed schemes.

3.1 Details of Profiling and Simulation Infrastructure

3.1.1 Profiling and Measurement Infrastructure on Real Machines

This dissertation validates the cloning accuracy of the system-level proxy benchmarks against the original applications on real systems by monitoring hardware performance counters. The interface for reading hardware performance counters involves accessing special CPU registers, which are called Model Specific Registers or MSRs as per the x86 terminology. Broadly, the MSRs can be categorized into two types. The first type of MSRs are called configuration registers and they are often used for starting or stopping the counters, setting up the interrupts for detecting overflows, choosing the events to monitor. The second type of MSRs are called counting registers which hold the counts of the chosen events. In general, somewhere between 2 to 8 counting registers are available on most contemporary computer systems. Accessing the configuration registers usually requires special privileged (ring 0, supervisor) instructions. Accessing counting registers may also require extra permissions. Linux's perf tool [53] provides a easy-to-use interface to access the processor performance counters. Similarly, power measurement on real systems is performed using the RAPL tools [41]. Intel's Pin tool [54] is used for performing micro-architecture independent workload characterization. Pin is a dynamic binary instrumentation tool and it can dynamically instrument the compiled binary files.

3.1.2 Simulation Infrastructure

3.1.2.1 System Performance Simulation

This dissertation uses MacSim, a cycle-level architecture simulator, to simulate the system-level performance of different applications on x86-based CPUs. MacSim models the architectural behavior of modern-day processors, including detailed pipeline stages (in-order and out-of-order) and the memory system components including caches, networks-on-chip, and memory controllers. MacSim can be used to simulate both homogeneous and heterogeneous ISA multicore simulations. To simulate x86 systems, MacSim uses a Pin tool as an instruction emulator, which feeds x86 instructions into the back-end simulation engine.

3.1.2.2 Memory Timing Simulation

This dissertation uses Ramulator, a memory system timing simulator, to evaluate the memory system performance in detail. Ramulator models the timing of the memory system quite accurately and supports a wide variety of commercial, as well as academic, DRAM standards such as DDR3/4, LPDDR3/4, GDDR5, WIO1/2, HBM, SALP, ALDRAM, TL-DRAM, RowClone, etc. Also, Ramulator is implemented in an efficient such that it decouples the work needed to interact with or query the core algorithm from the work needed to update its internal state-machines. As a result, integrating Ramulator with a processor simulation model does not slow down the processor simulation much. Ramulator comes with a simple memory controller module, which exposes an API for sending and receiving requests to and from the memory system. It supports two modes of operations: one mode for standalone usage and another for integrated usage with a processor/cache simulator. This dissertation uses Ramulator in both standalone and integrated modes in order to enable different experimental studies.

3.1.2.3 Cache Hierarchy Simulation

This dissertation uses CMP$im, a cache-hierarchy performance simulator to evaluate the proposed memory proxy benchmarking techniques across different cache hierarchy and prefetcher configurations. CMP$im uses Pin to serve as the functional model that provides CMP$im with memory addresses and other related information. CMP$im is quite configurable and can gather detailed cache performance statistics for single-core and multi-core configurations. CMP$im enables different types of cache-hierarchy studies where users can vary the cache parameters (e.g., cache size, associativity, cache line size), cache allocation policies, cache replacement policies, and write policies. The number of levels in the cache hierarchy is also configurable, along with the specification of an inclusion/exclusion and cache-sharing (applicable for chip multi-processors) policy. This dissertation augments the CMP$im infrastructure with different prefetcher modules in order to study the impact of prefetchers on cache performance. It also interfaces CMP$im with a detailed memory system simulator to evaluate the performance of the underlying memory system.

3.1.2.4 GPU Performance Simulation

To evaluate the cloning accuracy of the GPU proxies, this dissertation uses GPGPU-Sim [6], a detailed general-purpose GPU (GPGPU) simulator that models the compute architecture of modern NVIDIA GPUs. GPGPU-Sim executes applications compiled to PTX (NVIDIA's intermediate instruction-set) or disassembled native GPU machine code. GPGPU-Sim is a functional-first simulator; it first functionally executes all instructions and then feeds them into the timing simulator. GPGPU-Sim models the functional and timing components of the compute pipeline e.g., the thread scheduling logic, highly-banked register file, special function units. GPU applications can access different types of memories. For example, global memory is the main data-store where most data resides. The global memory data is cached in the on-chip multi-level cache hierarchy. Other GPU-specific memory types include constant memory (used for handling GPU read-only data), scratchpad memory (a software-managed on-chip cache used mostly for saving spilled registers), texture cache (graphics-specific cache), parameter cache (for storing compute kernel parameters) and instruction cache (for storing kernel's instructions). GPGPU-Sim includes models for all the different types of GPU memory as well as the on-chip caches and the DRAM memory system. GPGPU-Sim consumes mostly unmodified GPGPU source code that is linked to GPGPU-Sim's custom GPGPU runtime library. The modified runtime library intercepts all GPGPU-specific function calls and emulates their effects. When a compute kernel is launched, the GPGPU-Sim runtime library initializes the simulator and executes the kernel in timing simulation. The main simulation loop continues executing until the kernel has completed before returning control from the runtime library call.

3.2 Workload Description

3.2.1 Databases

This dissertation uses three modern, real-world databases (Cassandra, MondoDB and MySQL) for evaluating the effectiveness of the proxy benchmark generation techniques.

3.2.1.1 Cassandra

Apache Cassandra [16] is a popular, Java-based column-family style NoSQL database. It is incrementally scalable, eventually consistent, and has no single point of failure. Every node in the Cassandra cluster knows of and has the key for at least one other node and any node can service a request. The node structure can be visualized as a ring/web of interconnected nodes. Cassandra is semi-structured; i.e., its data may share some of the same fields or columns, but not all of them. In this way Cassandra is slightly more organized than MongoDB, but still not as rigid as MySQL.

3.2.1.2 MongoDB

MongoDB [56] is an open-source, C++ based document-style NoSQL database. It is designed for speed and scalability. It has a flexible schema (allows objects to not have fixed schema/type) and can store large documents such as binaries, images and audio files. Documents are stored as binary JSON objects and may be organized into collections. Within a collection each document has a primary key, and an index can be created for each query-able field. MongoDB's data is searched using keys and meta-data information.

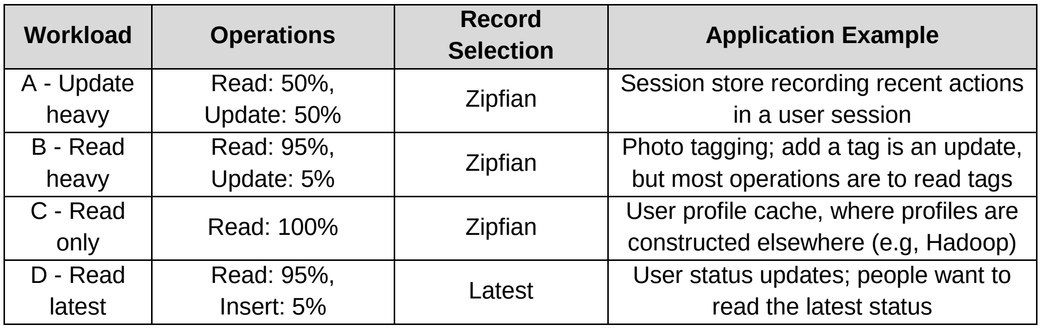
3.2.1.3 MySQL

MySQL [58] is one of the world's most popular open-source relational database management system. It enables the cost-effective delivery of reliable, high-performance and scalable web-based and embedded database applications. MySQL is designed to work on data whose fields are pre-defined and finite in number. Given this regular layout, MySQL can organize and search through data in multiple dimensions. This is both its strength and limitation, as it can't use the same strategy on less structured data.

3.2.2 Data-serving Benchmarks

This dissertation uses the Yahoo! Cloud Serving Benchmark (YCSB)[18] to represent the data-serving applications. YCSB is a standard benchmarking framework that is used to evaluate different cloud systems. YCSB's framework consists of a workload generating client and a set of standard 'core' workloads (see Table 3.1), which cover the most important operations performed against a typical data-serving database. The test database is generated using the YCSB framework and has over 10 million records (total size is $\geq$ 12GB). The data-set size is chosen so that the data fits into the memory of the server nodes, which is the recommended operational setup for scale-out applications for better performance [57]. Every test run performs 1 million operations against the database.

Table 3.1: Yahoo! Cloud Serving Benchmark (YCSB) Core Workloads



Workload	Operations	Record Selection	Application Example
A - Update heavy	Read: 50%, Update: 50%	Zipfian	Session store recording recent actions in a user session
B - Read heavy	Read: 95%, Update: 5%	Zipfian	Photo tagging; add a tag is an update, but most operations are to read tags
C - Read only	Read: 100%	Zipfian	User profile cache, where profiles are constructed elsewhere (e.g, Hadoop)
D - Read latest	Read: 95%, Insert: 5%	Latest	User status updates; people want to read the latest status

3.2.3 Data-analytics Benchmarks

This dissertation uses TPC-H benchmarks [100] to represent the data-analytics class of applications. TPC-H models a decision-support system environment for commercial order processing engines. It consists of a set of queries that interact with the server system to perform different business-like analyses. Similar to Barroso et al. [10], the experiments use a data-set size ($\sim$10GB) to analyze the behavior of an in-memory database. The dbgen and qgen tools (provided on TPC's website) are used to create/populate the database and generate the queries. Five different queries are evaluated from the TPC-H benchmark suite on MySQL database. Query details are shown in Table 3.2.

3.2.4 SPEC CPU2006 and SPEC CPU2017 Benchmarks

SPEC CPU2006and SPEC CPU2017 are a popular set of standard benchmarks that are widely used in the evaluation of the CPU performance. SPEC CPU2006 suite consists of 29 benchmarks, including integer and floating point

Table 3.2: TPC-H Benchmark Description

Sl.	Benchmark Name	Description
1	TPC-H Query 1 (Q1)	Pricing summary report query involving sequential table scan.
2	TPC-H Query 3 (Q3)	Shipping priority query, involves hash-join,nested loop join
3	TPC-H Query 6 (Q6)	Forecasting revenue change query using sort
4	TPC-H Query 14 (Q14)	Business Promotion Effect Query using join
5	TPC-H Query 19 (Q19)	Discounted revenue query using nested loop join

suites, ranging from CPU-intensive to memory-bound applications. SPEC CPU2006 workloads do not use extensive system I/O traffic and are single threaded. In order to use this suite on multi-core platforms, multi-programmed workloads are formed by running individual instances of the CPU2006 benchmarks on each core. SPEC CPU2017 suite consists of 43 benchmarks, including speed and rate sub-suites of floating-point and integer benchmarks. The rate benchmarks in the CPU2017 suite are multi-programmed workloads.

Unlike its predecessor, the SPEC CPU2017 suite is divided into four categories: speed integer (SPECspeed INT), rate integer (SPECrate INT), speed floating point (SPECspeed FP) and rate floating point (SPECrate FP). The SPECspeed INT, SPECspeed FP and SPECrate INT groups consist of 10 benchmarks each, while the SPECrate FP group consists of 13 benchmarks. The CPU2017 benchmarks are written in C, C++ and Fortran languages. Compared to the CPU2006 FP benchmarks, the CPU2017 FP benchmarks have $\sim$10X higher dynamic instruction count.

This steep increase in instruction counts further exacerbates the problem of benchmark simulation time on most state-of-the-art simulators.

3.2.5 Other Benchmarks

Memcached - Today's web applications are very latency-sensitive with strict quality-of-service (QoS) requirements. As these applications are highly data-intensive and have very big data footprints, they spend a significant fraction of their execution time servicing requests to the memory system. Disks or hard-drives are too slow to meet the QoS requirements of modern-day applications. As a result, most server systems use dedicated caching servers to cache data in their DRAM. Memcached benchmark [26] relies on one such widely-popular data-caching platform and simulates a Twitter caching server using a real Twitter dataset.

Graph Analytics - In contrast to the data-analytics benchmark that operates on textual data, the graph analytics benchmark [26] analyzes large scale graphs. Graph analytics is becoming increasing popular due to the growing popularity of social networks such as Facebook and Twitter. The Graph Analytics benchmark [26] uses the GraphLab machine learning and data mining software to run the TunkRank algorithm, which recursively computes the influence of Twitter users based on the number of their followers.

3.2.6 GPU Benchmarks

Rodinia - The Rodinia benchmarks [17] are designed to evaluate heterogeneous computing infrastructures. As they are based on OpenMP and CUDA, they can target both GPU architectures as well as multi-core CPU architectures. The Rodinia suite is structured to span a range of parallelism and compute patterns, providing researchers with various feature options to identify architectural bottlenecks and to fine tune hardware designs.

Parboil - The Parboil benchmarks [96] are a set of throughput computing applications useful for studying the performance of throughput computing architectures and compilers. The benchmarks include throughput computing applications in many different scientific and commercial fields including image processing, biomolecular simulation, fluid dynamics, and astronomy.

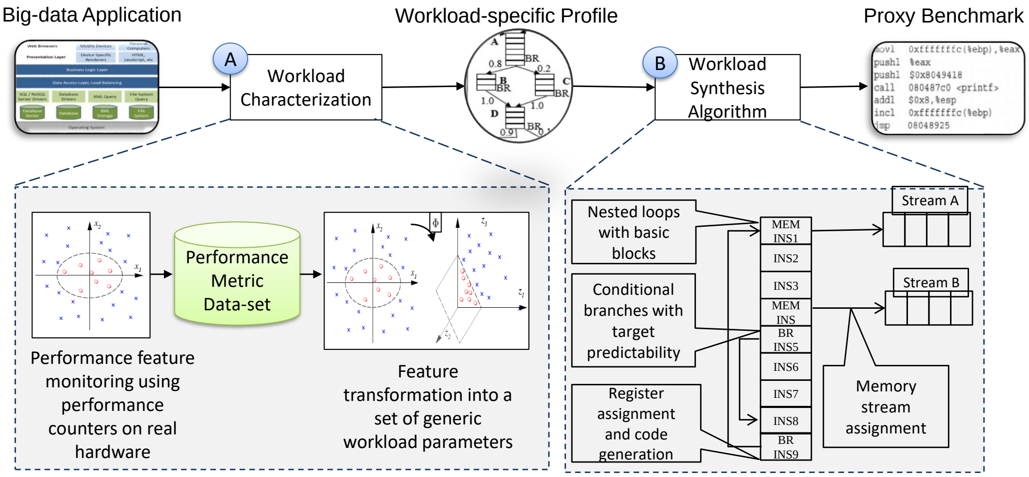
Chapter 4

PerfProx: Core-level Proxy Benchmarks for Emerging Workloads using Performance Counter Based Characterization

Prior proxy benchmark generation techniques [11, 43, 29] use micro-architecture independent metrics derived from detailed functional simulators for proxy generation. Such metrics are often very difficult to generate for many emerging applications. Several big-data workloads (e.g., Java-based Cassandra) do not work reliably with many detailed simulation models (e.g., gem5) or profiling tools (e.g., Pin [54], Valgrind, etc.). To overcome the challenges in simulating such complex emerging applications, this chapter proposes a proxy synthesis methodology, "**PerfProx**" [71, 69], which uses performance metrics derived primarily from hardware performance counter measurements to synthesize miniature proxy benchmarks. The generated proxy benchmarks are representative of the performance of original applications and yet, converge to results quickly without any complex software-stack support. Several big-data workloads do not work reliably with many profiling tools, but performance-counter based characterization and associated extrapolation into generic parameters that the code generator can enable fast and efficient proxy generation for such benchmarks.

4.1 PerfProx's Methodology

In this section, the proxy generation methodology of PerfProx is described. Figure 4.1 shows PerfProx's overall framework. PerfProx first characterizes the database applications running on real hardware and extracts their key performance metrics (step Ⓐ). During the workload characterization step, PerfProx captures low-level dynamic runtime characteristics of the program (like statistical simulation), continuously building accurate instruction-locality, memory access and branching models. Based on the extracted performance features, PerfProx builds a ***workload-specific profile*** for each database application that uniquely summarizes the application's runtime behavior over its entire execution time (step Ⓑ). Synthesizing using statistics rather than the original application source code effectively hides the functional meaning of the code/data, which addresses any proprietariness or con-


Big-data Application
A
Workload Characterization
Workload-specific Profile
A
BR
0.8
0.2
B
BR
C
BR
1.0
1.0
D
BR
0.9
0.1
B
Workload Synthesis Algorithm
Proxy Benchmark
Performance feature monitoring using performance counters on real hardware
Performance Metric Data-set
x_1
x_2
z_1
z_2
z_3
Φ
Feature transformation into a set of generic workload parameters
Nested loops with basic blocks
Conditional branches with target predictability
Register assignment and code generation
MEM INS1
INS2
INS3
MEM INS
BR INS5
INS6
INS7
INS8
BR INS9
Stream A
Stream B
Memory stream assignment

```
movl   0xfffffffc(%ebp),%eax
pushl  %eax
pushl  $0x8049418
call   080487c0 <printf>
addl   $0x8,%esp
incl   0xfffffffc(%ebp)
jmp    08048925
```

Figure 4.1: PerfProx's Proxy Generation Methodology

fidentiality concerns about sharing end-user workloads. Finally, PerfProx's workload synthesizer uses the captured workload-specific profiles to generate the proxy benchmarks, which have similar features as the original application (step ©).

If the workload-specific profile represents the execution behavior of the original application accurately, then the proxy benchmark created using the same set of features should also replicate the performance of the original applications with similar accuracy. The proxy benchmark is synthesized as a C-based program, with low-level instructions instantiated as *asm* statements. When compiled and executed, the proxy benchmark mimics the dynamic performance characteristics of the database application and it can be easily run on early performance/functional simulators, etc. with significantly reduced runtimes. The following paragraphs will describe the workload characterization methodology and the proxy synthesis algorithm in detail.

4.1.1 Workload Characterization using Performance Counters

As discussed, PerfProx monitors the runtime behavior of an application and produces a set of workload characteristics representing its low-level dynamic execution characteristics. PerfProx captures the execution characteristics of database applications primarily using hardware performance counters running on real hardware systems. It then transforms the performance counter data using analytical models to derive features representing the workload-specific profile. The workload-specific profile serves as an input to the workload synthesis algorithm, which generates representative proxy benchmarks that closely resemble the performance of the original applications. Many emerging big-data workloads do not work reliably with many profiling tools, and thus performance-counter based characterization and associated extrapolation enables fast and accurate analysis and proxy generation for such applications. For database applications that can work with fast program profilers (e.g., Pin), PerfProx further augments its memory access modeling methodology by capturing micro-architecture independent patterns from the original memory access streams. The key performance features captured by the PerfProx's workload characterization model are described in the following sections. The abstract workload-specific profile generated based on the following features is shown in Table 4.1.

a. Instruction Mix The instruction mix (IMIX) of a program measures the relative frequency of various operations performed by the program. PerfProx measures the instruction mix of the database applications using hardware performance counters. PerfProx specifically measures the fraction of integer arithmetic, integer multiplication, integer division, floating-point operations, SIMD operations, loads, stores and control instructions in the dynamic instruction stream of the program. The detailed instruction mix categorization is shown in Table 4.1. PerfProx computes the target proxy IMIX based on the fraction of individual instruction types in the original application. This target IMIX fraction is used to populate corresponding instructions into the static basic blocks of the proxy benchmark.

b. Instruction Count and Basic Block Characteristics PerfProx uses the database application's instruction cache (icache) miss rate to derive an initial estimate of the

Table 4.1: PerfProx's Workload-specific Profile

Metric Category	Metrics	Description/Range
Instruction-mix	1. Load, 2. Store, 3. Integer 4. INT MUL, 5. INT DIV, 6. FP, 7. SIMD, 8. Control instructions	Fraction of each instruction category measured using hardware performance counters
Instruction Footprint	9. Instruction count	Derived from target instruction cache miss rate and default cache configuration assumption
Control-flow Predictability	10 . Branch transition probability	Derived from target branch misprediction rate (Ranges between 0-100%)
	11. Average basic block size	Derived from actual application's total instruction count and control instruction count
	12. Number of basic blocks	Derived from metrics 9 and 11
Instruction-level Parallelism	13. Instruction dependency distance	1, 4, 8, 16, 32, 64, 128, 256 dependency distance bins
Memory Access Model	14. Stride value per static load/store	0, 1, 2, 4, 8, 12, 16, 20, 24, 28, 32, 64, 128, 256 byte buckets based on target L1/L2 cache miss rate or characterization of application's local and global strides.
	15. Data footprint (number of iterations before resetting to beginning of data arrays)	Based on target application data footprint
	16. Memory stream concurrency factor	Bins representing upto 100 different data arrays
System Activity	17. System call ratio	Derived from target fraction of user vs kernel instructions

number of instructions to instantiate in the proxy benchmark. The instruction cache miss rate metric is easily measurable on most computers using the hardware performance counters that count the number of instruction cache misses and accesses. An initial estimate of the number of static instructions to instantiate in the proxy benchmark is made to achieve the desired icache miss rate based on the assumption of a default instruction cache size/configuration (64KB, 64B line-size, 2-way set-associative). The final static instruction count of the proxy benchmark is tuned to achieve the target icache miss rate on the profiled hardware system. PerfProx also measures the average basic block size of the database application based on its total dynamic instruction count and fraction of control instructions. Both these metrics are measured using hardware performance counters on the profiled system. The number of static basic blocks to instantiate in the proxy benchmark is derived as a ratio of the final instruction count estimate and the target basic block size.

c. Control Flow Behavior Another important metric that affects application performance significantly is its control flow performance. PerfProx estimates the overall branch predictability of an application in a directly correlated fashion based on the application's branch misprediction rate (measured using hardware performance counters). To model a target branch predictability into the proxy benchmark during proxy generation, PerfProx estimates the fraction of control instructions in the proxy benchmark that will have a particular predictability behavior. For example, assuming a 2-bit saturating counter based predictor, 100% and 50% branch predictability can be modeled using a branch instruction which is mostly not-taken and a branch instruction which alternates between the taken and not-taken paths respectively. Similarly, a very hard-to-predict branch can be modeled to switch between the taken and not-taken paths in a random fashion.

d. Memory-access Model Although PerfProx's primary objective is to develop a fast and light-weight methodology to model application performance, it is crucial for PerfProx to model the cache and memory performance accurately. The principle of data locality and its impact on cache and memory performance is widely recognized. PerfProx models the data memory accesses using simple, strided stream-classes over fixed-size data arrays. PerfProx leverages a methodology to infer the memory stream strides based on the data cache miss rates of the original application (similar approach as [11, 25]). It employs a pre-computed table that holds the correlation between L1/L2 cache hit rates and the corresponding stride values. Particular memory access strides are determined, using the target L1/L2 hit rate information along with this table, by first matching the L1 hit rate of the memory operation, followed by the L2 hit rate. For example, memory accesses with 100% hit-rate can be modeled using a stride of zero (assuming 64B cacheline size). Stride values are optimized to achieve the highest correlation of proxies in terms of target cache performance. Although approximate, such a mechanism to model strides based on the cache miss rates enables fast and efficient memory pattern modeling of complex workloads, which are otherwise difficult to simulate on detailed performance simulators. This technique was used to estimate memory access strides for database applications (e.g., JAVA-based Cassandra) that often can not reliably run to completion using program profilers.

Despite its advantages, the simple memory access model based on cache miss rates is dependent on the profiled cache/memory hierarchy. Although it is possible to measure cache miss rates corresponding to different cache sizes/configurations in a single run, a better solution for improving the fidelity of the generated proxies would be to exploit micro-architecture independent features to model the memory access locality. Thus to improve upon its memory access locality modeling technique, PerfProx proposes to analyze detailed access patterns in the global and local memory access streams of the database applications. More specifically, memory access behavior is modeled by finding fine-grained stride-based correlations on (a) per-instruction (local-stride profile) and (b) global memory reference stream (global-stride profile) granularity. The collected stride information is categorized into bins, where each bin corresponds to a stride between -256 to +256 for global strides and 0 to 2^{18} for local strides. During proxy generation, every proxy memory instruction is assigned a memory address that satisfies both the target local and global stride distribution of the original application. This methodology can also model irregular memory access behavior by controlling the degree of spatial locality in memory streams and randomly using large stride bins. For database applications (e.g., MySQL applications) that can work with fast program profilers, the local and global memory strides were measured at a byte-size granularity over the entire execution period.

Furthermore, it has been shown by several prior research studies that database applications tend to have higher TLB misses (often as frequent as cache misses) [67, 26], which has a significant impact on their performance. As discussed before, PerfProx models the data memory accesses using simple strided stream classes over fixed-size data arrays. In order to model the effects of TLB performance, PerfProx controls the degree of concurrency in its active memory access streams; i.e., it controls the number of unique memory streams actively accessed by the proxy application within a fixed window of instructions. Individual load/store instructions are assigned to different active data streams based on this concurrency factor. The proxy data footprint is also scaled according to the target data-set size of the database application.

e. Instruction-level Parallelism Instruction-level parallelism (ILP) is an important determinant of application performance. Tight producer-consumer chains can limit performance due to serialization effects. PerfProx models the original application's ILP based on its inter-instruction dependency distance, which is defined as the number of dynamic instructions between the production(write) and consumption(read) of a register/memory operand. PerfProx classifies the instruction dependency distance into 8 bins, where each bin represents the percentage of instructions having that particular dependency relation. As it is not possible to measure the application's exact dependency distance using performance counters alone, PerfProx adopts an approximate model to measure the same. It makes an initial estimate of the application's inter-instruction dependency using the dependency-related stall events of the original application. Most micro-architectures support measuring some form of reservation-station stalls, re-order buffer stalls, or data-dependency stalls, etc. Depending on the ratio of the dependency related stalls to overall execution cycles, ranging from very low ($\leq 2\%$) to high($\geq 30\%$), PerfProx approximately extrapolates the inter-instruction dependencies into the 8 bins (see Table 4.1), where each bin represents a certain inter-instruction dependency distance. The final dependency distance estimate is tuned to achieve the target stall ratio on the profiled system. Nonetheless, profiling the original applications to measure the exact multi-granularity instruction dependency distance statistics (if possible) can lead to more accurate modeling of instruction-level parallelism effects. During proxy benchmark generation, the register/memory operands of the instructions are assigned a dependency distance to satisfy the metrics collected from the original application.

f. System Activity Prior research [67, 26] has shown that the emerging database applications spend a significant fraction of their execution time executing operating system (OS) code, which has a significant impact on their overall performance. To model the performance impact of high system activity, PerfProx measures the system activity in the original applications using the *STRACE* tool and the fraction of executed user-mode and kernel instructions using hardware performance counters. During proxy generation, PerfProx inserts corresponding desired fraction of system calls into the basic blocks in the proxy benchmark to achieve the desired level of system activity.

4.1.2 Synthetic Proxy Benchmark Generation

Algorithm 1 PerfProx's Workload Synthesis Algorithm

```
 1: Input: Table 4.1 metrics, target instruction & basic block count;
 2: Output: Proxy benchmark sequence, B[]
 3: Determine number of static basic blocks $B$ to instantiate in proxy benchmark.
 4: while $b < B$ do
 5:     Sample a random basic block.
 6:     Estimate basic block size $I$ to satisfy mean & std. dev of target basic block size.
 7:     for $i < I$ do
 8:         Assign instruction type based on target IMIX probability.
 9:         Assign dependency relation based on target dependency distance distribution.
10:         For load/store instructions, assign the memory access stream and local stride.
11:         Inject system-calls based on target system-call frequency.
12:         Insert x86 test operation with chosen modulo operand.
13:         Assign last instruction to be conditional branch instruction.
14:     end for
15: end while
16: Assign architectural register operands to satisfy dependency relations of step 9.
17: return B[]
```

In this section, the workload synthesis algorithm (see Figure 4.1) is discussed in detail. The workload synthesis algorithm takes as an input the workload-specific profile captured during the workload characterization phase (see Table 4.1 for a list of the captured metrics). The proxy benchmark generation steps are listed in Algorithm 1. PerfProx first estimates the total number of static basic blocks to instantiate in the proxy benchmark. It then chooses a random number in the interval [0, 1] to select a basic block based on its access frequency. The size of the basic block (in terms of number of instructions) is chosen to satisfy the mean and standard deviation of the target basic block size (line 6). The IMIX statistics are used to populate the basic block with appropriate instruction types (line 8), while ensuring that the last instruction of every basic block is a conditional branch instruction (line 13). Every instruction is assigned a dependency distance (i.e., a previous instruction that generates its data operand) to satisfy the dependency distance criterion. The memory instructions are assigned a stride and memory access stream based on the memory model described before. System calls are injected (or not) into the basic block based on the target system-call frequency. Finally, an X86 test operation is inserted before the branch instruction to set the condition codes that affect the conditional branch outcome. The test instruction's operand is chosen to control the branch transition rate in order to satisfy the target transition rate of every basic block. These steps are repeated till the target number of static basic blocks are generated. Finally, architectural register operands are assigned to each instruction to satisfy the dependencies in step 9 (line 16).

The proxy synthesizer generates C-language based proxy benchmarks with embedded X86-based assembly instructions using the *asm* construct. The generated sequence of instructions is nested under a two-level loop where the inner loop iterations controls the dynamic data footprint and the outer loop iterations control the number of dynamic instructions in the proxy benchmark. The nested looping structure is not the major determinant of the application performance as the static footprint of the proxy benchmarks is significant. As an example, the proxy benchmark of YCSB workload with MongoDB consists of over 40K static basic blocks. The outer loop iterations reset each data-stream access to the first element of the memory array (for re-walking). The code is encompassed inside a main header and the malloc library call is used to statically allocate memory for the data streams. Using the *volatile* directive for each *asm* statement prevents the compiler from optimizing out the program machine instructions.

4.1.3 Discussion

PerfProx's workload characterization methodology has several advantages. One of its key benefits is speed. As PerfProx derives key workload metrics from hardware performance counters using simple models, PerfProx can run at the speed of native hardware. Thus, PerfProx makes it possible to monitor complex, long-running applications over their entire execution time, which is often impossible on slower, detailed performance models. Also, many database applications (e.g., Cassandra) are difficult to run reliably using performance simulators as they are based on higher level programming languages such as JAVA and typically require deep software stack support. PerfProx provides an easy and reliable methodology to evaluate such applications and generate corresponding proxy benchmarks. For applications that can work with fast program profilers (e.g., Pin), PerfProx also augments its memory access modeling methodology by capturing micro-architecture independent patterns from the original memory access streams to improve fidelity of the generated proxies. Nonetheless, the reliance on some micro-architecture dependent features for proxy generation can degrade the performance correlation of the PerfProx proxies on systems that deviate significantly from the target system (which was used for performance counter based profiling and proxy generation).

It must be noted that the data-set and query information manifest themselves into the final workload characteristics obtained from the dynamic statistical profiling of the application. Separate proxy benchmarks need to be generated for representing different input data-sets and database application queries; however, the fast proxy benchmark synthesis methodology makes this feasible. Also, the generated proxies do not capture features that are not modeled (e.g., value prediction) during the workload characterization step. Also, PerfProx models the behavior of in-memory databases and thus, does not model I/O effects. This is not an inherent limitation of the approach as support could be added by monitoring/modeling I/O (beyond the scope of this paper). PerfProx also does not model context-switches and applications are pinned to cores during execution.

4.2 Evaluation

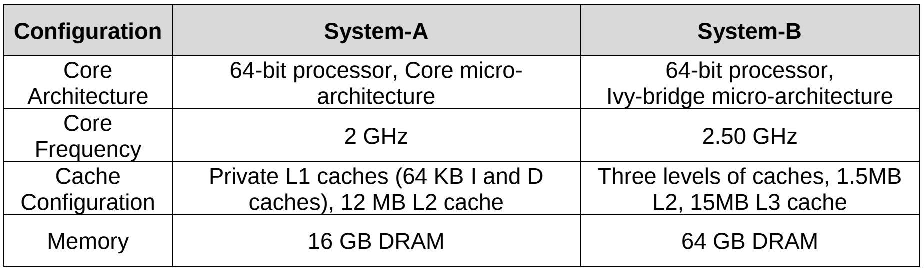
This section discusses the experimental setup followed by a detailed evaluation of PerfProx's performance cloning accuracy.

4.2.1 Experimental Setup

PerfProx is evaluated using three NoSQL and SQL databases: Cassandra (version 0.1.7), MongoDB (version 2.6.5) and MySQL(version 5.1.15). MongoDB is setup to run one mongod instance per server node. MongoDB's config server and router node are setup on the server node, and it was verified that the router node and config server processes are light-weight and are not bottlenecks in the performance tests. Cassandra database is setup and run using Java Oracle JDK version 1.7, with a JVM heap size of 8GB. Yahoo! Cloud Serving Benchmarks (YCSB)[18] are used to represent the data-serving applications using Cassandra, MongoDB and MySQL databases. TPC-H benchmarks [100] are used to represent the data-analytics applications.

Characterization and generation of proxy benchmarks for databases running YCSB and TPC-H workloads is performed on servers based on the system-A configuration, as described in Table 4.2. The performance of PerfProx proxies is validated on systems A and B, as shown in the table.

Table 4.2: Systems used for Evaluating PerfProx's Cloning Accuracy



Configuration	System-A	System-B
Core Architecture	64-bit processor, Core micro-architecture	64-bit processor, Ivy-bridge micro-architecture
Core Frequency	2 GHz	2.50 GHz
Cache Configuration	Private L1 caches (64 KB I and D caches), 12 MB L2 cache	Three levels of caches, 1.5MB L2, 15MB L3 cache
Memory	16 GB DRAM	64 GB DRAM

The proxy benchmarks are compiled using gcc with the -O0 optimization flag to avoid compiler optimizations that remove dead-code or alter the inserted code in other ways. In order to profile the original applications and evaluate the microarchitectural performance of the actual applications and corresponding proxy benchmarks, Linux's perf tool [53] is used to provide an interface to the processor performance counters. Intel's PIN tool [54] is also used for workload characterization.

4.2.2 Results and Analysis

This section extensively evaluates the proxy benchmarks to see how well they can replicate the behavior of the original database applications in terms of key performance metrics across different systems. In the following sections, YCSB benchmarks are represented as DB-WLx, where DB is the original database name and x is the YCSB workload (A-D). Also, database and proxy benchmark results are represented as Actual (A) and Proxy (P) respectively. Apart from comparing the percentage error between different performance metrics of the proxy and database applications, the Pearson's correlation coefficient (ρ) is also reported. Pearson's correlation coefficient indicates how well the proxy benchmarks track the trends in the actual database applications, with 1 indicating a perfect correlation, and 0 indicating no correlation.

4.2.2.1 Performance Validation of Proxy Benchmarks

Figure 4.2 compares the instructions per cycle (IPC) of Cassandra, MySQL and MongoDB databases running the YCSB and TPC-H benchmarks along with their corresponding proxies on system-A. It can be observed that the IPC of the proxy benchmarks closely follows the IPC of the original applications, with a high correlation between the two ($\rho = 0.99$). The mean error between the proxy IPC and actual application IPC is 6.1% approximately (10.7% max) across all workloads. Considering the data-serving applications only, the average error in IPC between the proxy and the actual applications is 5.1%. MongoDB experiences worse errors as compared to Cassandra and MySQL. Performance of MongoDB-based applications are impacted by their cache and TLB performance [67]. Because of PerfProx's simple memory access locality modeling technique, PerfProx proxies experience higher deviation in terms of the cache and TLB performance with respect to the original applications, which results in the higher overall performance modeling error. The data-analytics applications have an average error of 6.5% between the proxy and actual applications.

Figure 4.3a compares the branch prediction rates of the original and their corresponding proxy benchmarks. It can be observed that the error between the

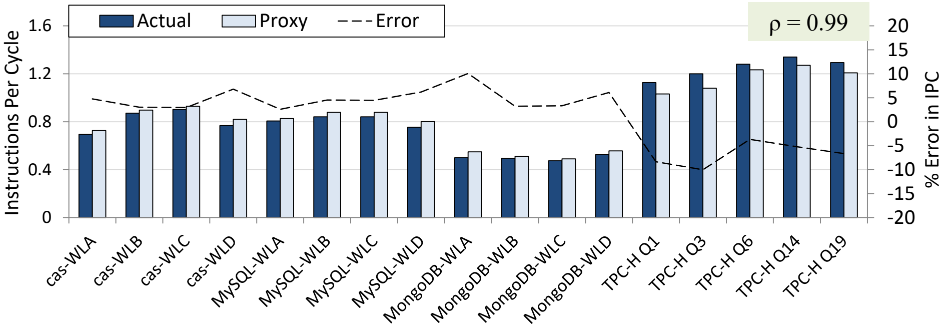


Figure 4.2: IPC of Real Databases and Proxy Applications on System-A

branch prediction rates of the proxy benchmarks and the actual applications is small (average error = 1.5%, ρ = 0.99). This shows that PerfProx's methodology of capturing and mimicking branch transition rates is effective at achieving the target branch prediction rates fairly accurately. Figures 4.3b and 4.3c compare the L1 cache and last-level cache (LLC) hit rates respectively for Cassandra, MySQL and MongoDB databases running the YCSB and TPC-H benchmarks with their corresponding proxies (normalized with respect to the cache hit rate of Cassandra running YCSB WLA benchmark). The average error in mimicking L1 and LLC Cache

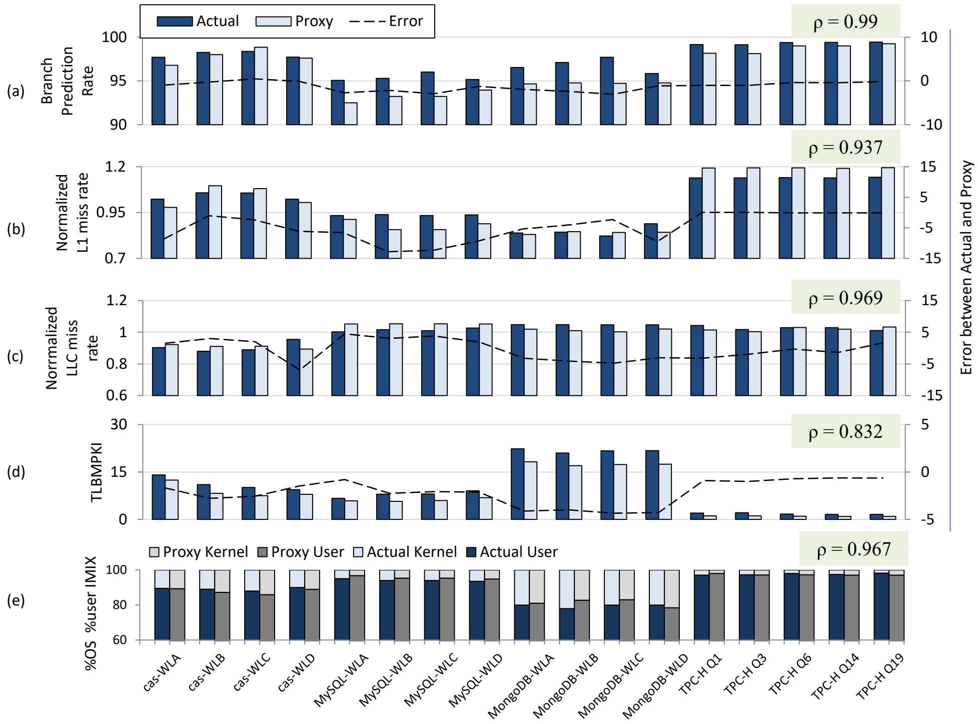


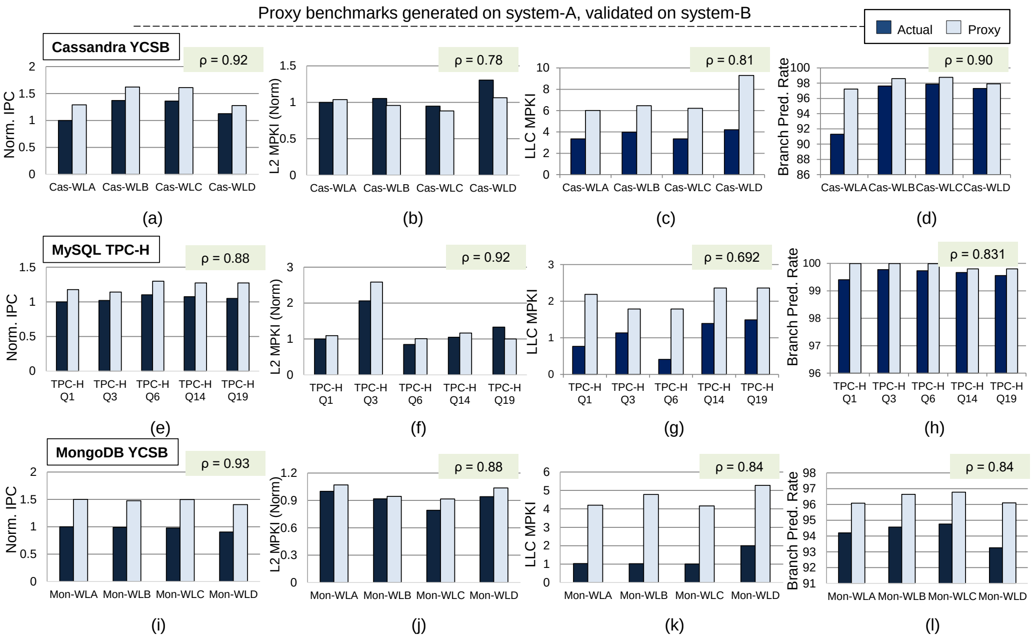
Figure 4.3: Comparison of Performance Features of Original and Proxy Applications on System-A. Error % on the Right-side Axis.

hit rate is 6.1% and 3.1% respectively. In terms of TLB behavior (Figure 4.3d), the average error between the proxy and original applications is higher as compared to other performance metrics. Nonetheless, the trend in TLB performance is captured to a reasonable degree across the different workloads (ρ = 0.83). Similarly, in terms of system activity (Figure 4.3e), the fraction of user to system instructions in the proxy benchmarks closely follows the original applications, with a correlation (ρ) of 0.967.

4.2.2.2 Proxy Cross-platform Validation

Next, the performance correlation of proxy benchmarks generated from system-A is validated on the system-B micro-architecture (see Table 4.2b).

Figure 4.4a shows the IPC of the proxy versus actual applications on system-B for Cassandra-based applications (normalized with respect to actual Cas-WLA). The proxy benchmarks experience an average error of ∼19.4% in replicating the IPC of the original applications across the different YCSB workloads. As PerfProx's workload features are derived using microarchitecture-dependent characterization (e.g., cache miss rates etc) based on a target system, the performance correlation of the proxies on similar machines is higher. However, when tested on machines with very different configurations, the performance correlation of proxies degrades. The original and proxy workloads are also compared using several other key metrics, e.g., L2 misses per kilo instructions (MPKI), LLC MPKI and branch prediction rate (see Figures 4.4b, 4.4c and 4.4d). It can be observed that, although the L2 and LLC MPKI of the proxy benchmarks follow the performance trends of



(a) (b) (c) (d)

(e) (f) (g) (h)

(i) (j) (k) (l)

Figure 4.4: Proxies from System-A Validated on System-B: (a) IPC, (b) L2 MPKI, (c) LLC MPKI, (d) Branch Prediction Rate

the original applications, the degree of correlation is lower because of the dependence of the profiled cache performance metrics on the profiled cache hierarchy.

Figure 4.4e compares the IPC of the proxy benchmarks versus the original TPC-H applications on system-B. The TPC-H proxy and original workloads are also compared across several other key metrics: L2 MPKI (average error = ∼0.78 MPKI), LLC MPKI (average error = ∼1 MPKI) and branch prediction rate (average error = 0.25%) in Figures 4.4f, 4.4g and 4.4h respectively. The L1 cache and TLB performance (not shown here due to space considerations) also have similar correlations between the original and proxy applications. Memory locality modeling using stride-based patterns leads to accurate capture of application spatial locality, thereby improving the cache performance correlation between the proxy and actual applications. However, the assumed stride model fails to capture long-distance reuse locality of accesses, which manifests as slightly worse errors in modeling LLC locality. On the other hand, the cache miss rate based memory locality modeling technique captures reuse probability at lower level caches as well, leading to slightly better performance correlation. Future work will focus on incorporating longer-distance reuse locality patterns into the memory access model.

Finally, Figure 4.4i shows the IPC of the proxy benchmarks versus the original applications on system-B for MongoDB-based applications. Although the average error between the IPC of proxy benchmarks and the original database queries on system-B is high, the proxy benchmarks still capture the IPC trends of the original application pretty well (average correlation = $\sim$0.93). The proxy and original workloads are also compared across several other key metrics: L2 MPKI, LLC MPKI (average error = $\sim$3 MPKI) and branch prediction rate (average error = 2.3%) as shown in Figures 4.4j, 4.4k and 4.4l respectively.

4.2.2.3 Proxy performance sensitivity analysis on different cache/TLB configurations

This section discusses the performance sensitivity of the proxy benchmarks to different cache and TLB configurations and aims to evaluate the effectiveness of PerfProx's memory access modeling methodology to capture and mimic the inherent memory access patterns in a workload.

First, a data-analytics application represented by the TPC-H Q19 benchmark is analyzed to evaluate its performance sensitivity. TPC-H Q19 is chosen because Q19's proxy experienced the highest error in replicating cache performance among the 5 TPC-H queries on system-A. A PIN-based cache simulator is used to measure the cache performance of the proxy and the original TPC-H queries across 20 different cache configurations, where the cache size and associativity are varied between 16-256KB and 1-32 respectively. Figure 4.5 shows the cache MPKI of the original TPC-H Q19 and its proxy for the different configurations. The cache MPKI of the proxy benchmark follows the original application with an average deviation of 0.5 MPKI and a high correlation of 0.89 across the different configurations.

Next, the cache performance sensitivity of data-serving applications is evaluated using the YCSB workload with MongoDB database. For testing different cache configurations, the cache size is changed between 16 to 256KB and associativity is changed between 1-16. Similarly, the different TLB configurations correspond to different TLB sizes (32-256 entries) and associativity (2-8). Figure 4.6

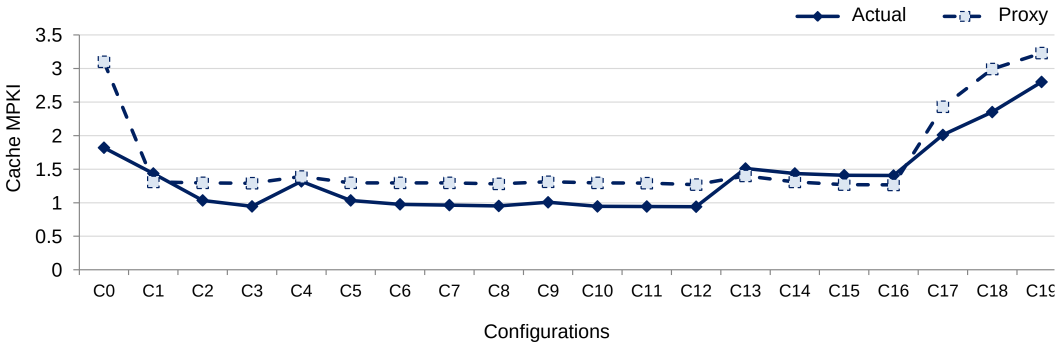


Figure 4.5: Performance Sensitivity of Data-analytics (TPC-H Q19) Proxy to Different Cache Configurations

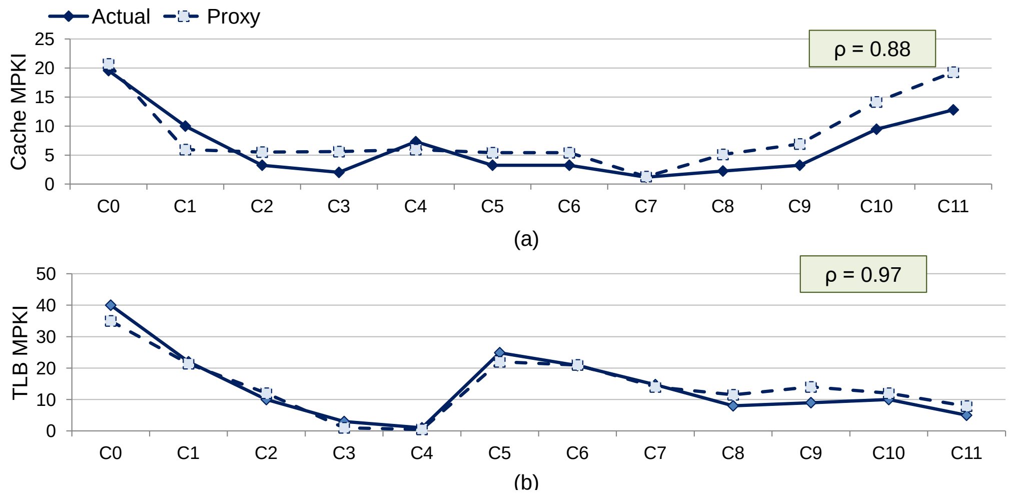


Figure 4.6: Sensitivity of Data-serving Proxy Performance to Different Cache and TLB Configurations

shows the cache and TLB MPKI sensitivity of the proxy and database application. It can be observed that the cache MPKI of the proxy benchmark follows the original application closely across different cache configurations, with an average deviation of ∼2 MPKI. Similarly, the TLB MPKI of the proxy benchmarks follows the original application with a mean error of 2.2 MPKI. The proxies' cache and TLB performance as compared to the actual applications have correlations of 0.88 and 0.97.

and the actual applications have a correlation of 0.88 and 0.97 with respect to cache and TLB performance respectively.

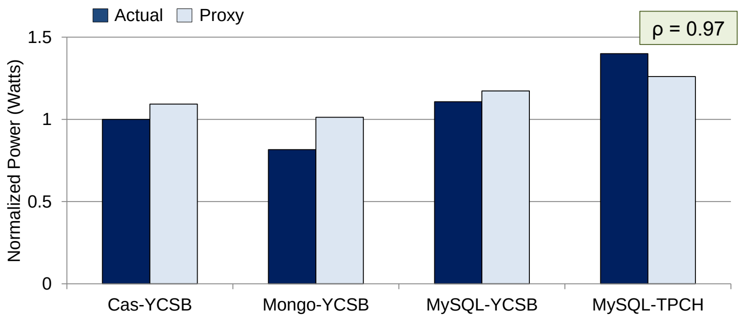


Figure 4.7: Comparing Power Consumption of Proxy versus Actual Applications

4.2.2.4 Energy-efficiency Analysis

Figure 4.7 shows the average power consumption (in watts) of the individual databases running the data-serving and data-analytics applications and their corresponding proxy benchmarks (normalized with respect to the actual YCSB benchmarks running on Cassandra). Power is measured using Intel's RAPL counters on system-B. There is a high degree of correlation between the average power consumption of the proxy and actual applications (ρ = 0.97). The power consumption of an application is often highly correlated with its performance behavior [28]. Since the proxy benchmarks mimic the performance behavior of the original applications closely in terms of IMIX, instruction dependencies, cache/memory behavior, they closely mimic the power characteristics of the actual applications as well.

4.2.3 Comparison with standard benchmarking suites

In this section, database applications are compared against three standard benchmark suites, SPEC CPU2006 [93], SPECjbb2013 [95] and Linpack [36]. The kiviat plots shown in Figure 4.8a show the performance trends of the original database applications and their corresponding proxy benchmarks, while the kiviat plots in Figure 4.8b shows performance metrics corresponding to the standard benchmarks across several key metrics. Specifically, the kiviat plots are based on selected raw performance metrics: L1D, L1I, LLC, I/D TLB MPKI, %kernel instructions executed (Ker), branch misprediction rate (BrMis), normalized by their maximum observed values on system-A.

Modern database applications suffer from several bottlenecks which limit their overall performance on contemporary hardware systems. The plots illustrate significant diversity in the performance and bottlenecks of different database applications and standard benchmarks. For example, SPECjbb stresses a different set of system components (branch misprediction rate and LLC cache misses) than MongoDB applications. Even with a comparable data-set size (over 10GB), Linpack does not encounter similar memory subsystem issues as the database applications, demonstrating that the Linpack program's behavior is different from databases even when the data-set is big. The plots also show how closely the generated proxy benchmarks resemble performance trends of the original workloads. Thus, the proxy benchmarks can be used for effective performance validation, while being very simple targets for performance evaluation. Improving the memory and instruction locality models can further improve their fidelity.

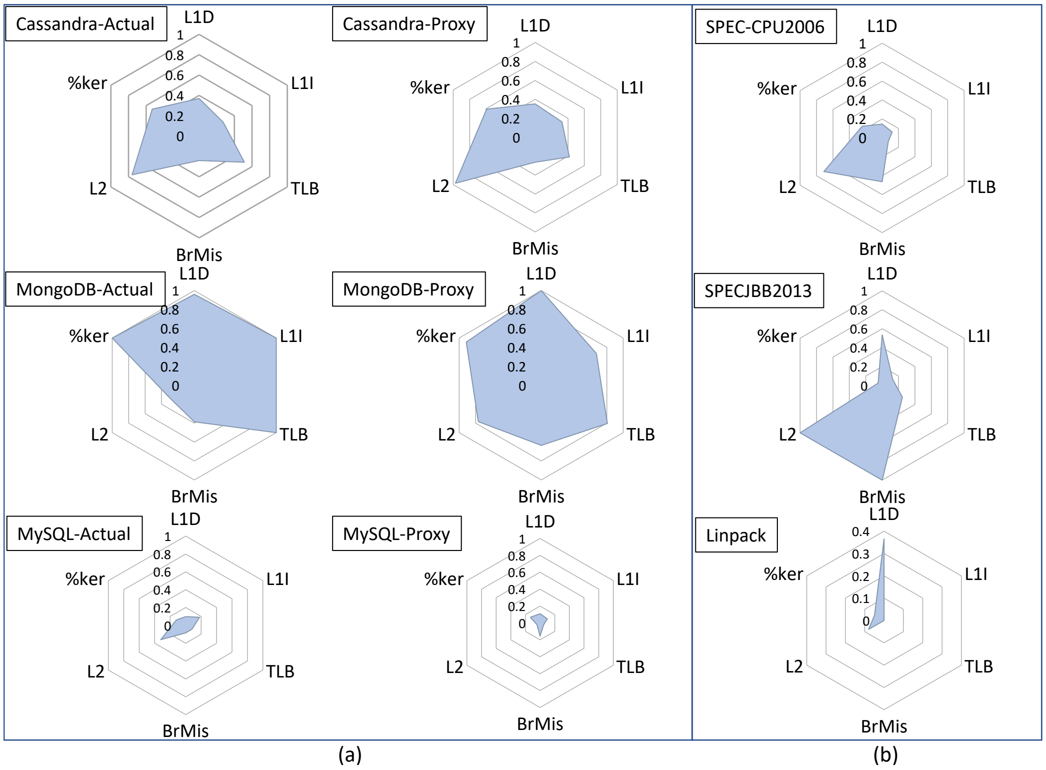


Figure 4.8: Kiviat Diagrams Comparing Performance of Original Database Applications, Proxy Benchmarks and a Set of Standard Benchmarks

4.2.3.1 Degree of Miniaturization

A key advantage of the proxy benchmarks is that they are miniaturized (have fewer instructions) as compared to the original applications. This significantly reduces the simulation time of the proxy benchmarks on simulation frameworks. The average instruction-count of the generated proxy benchmarks is ∼2 billion (∼520 times smaller than original database applications). Thus, the proxy-benchmarks can be run to completion on simulators in a reasonable time.

4.3 Summary

To simplify the benchmarking of emerging big-data applications, this chapter presented a novel methodology (PerfProx) to create representative system-level proxy benchmarks. PerfProx generates proxies by monitoring and extrapolating database application performance primarily using hardware performance counters. PerfProx proxies enable fast and efficient performance evaluation of emerging workloads without needing back-end database or complex software stack support. PerfProx is evaluated using three popular and modern databases, Cassandra, MySQL and MongoDB for both data-serving and data-analytics applications running across different hardware platforms and multiple cache/TLB configurations. The proxy benchmarks mimic the performance (IPC) of the original applications with 94.9% (average) accuracy for data-serving applications and 93.5% (average) accuracy for data-analytics applications, while significantly reducing the instruction counts compared to the original applications'.

Chapter 5

HALO: A Hierarchical Memory Access Locality Modeling Technique For Memory System Exploration

Memory system performance is a fundamental performance and energy bottleneck in all computing systems. Optimal design of memory system hierarchy requires an in-depth understanding of target end-user workload demands and extensive design-space exploration. To enable fast and efficient memory system design space exploration, this chapter presents a novel spatio-temporal model of end-user memory access streams, which can be used to explore memory-system trade-offs. Memory access streams of applications are shaped by several different factors - high-level algorithms that access different data-structures in the program, memory layout of data-structures determined by the compiler/runtime, program's unique control-flow, machine's execution model (out-of-order versus in-order execution, memory address reordering), etc. As a result, distilling the inherent patterns in the memory access streams into a small set of statistics is a very challenging problem.

As discussed in Chapter 2, prior memory locality modeling proposals [7, 5] mimic cache and memory behavior by tracking temporal or spatial locality patterns within the global memory reference streams. For example, WEST models only temporal locality patterns using per-set stack distance distributions and is thus, inadequate to evaluate microarchitectural structures that exploit spatial locality (e.g. prefetchers). STM models spatial locality by capturing global stride-based correlations in the memory reference stream. However, it has to maintain significantly long stride history-base tables to capture the dominant stride transitions, which results in significantly higher meta-data storage overhead. Limiting the stride history depth reduces storage overhead, but results in poor cloning accuracy.

The global access statistics are often not effective in capturing memory reference behavior because accesses to different structures are often interleaved, which mask the patterns within each individual stream. This can be illustrated with an example shown in Figure 5.1a. This simple program adds two array data-structures ($a[64]$ and $b[64]$), leading to a memory reference and stride pattern sequence shown in Figure 5.1b (assuming, 1 array entry = 1 byte = 1 cache-block). It can be observed that the global stride patterns are non-repetitive. Still, capturing the global stride sequence is feasible even with a 1-length global stride history table (see Figure 5.1e), but it would require saving every individual stride transition, which is almost equivalent to saving the entire memory trace. However, it can also be observed that accesses to the individual data-structures have significant regularity (+1 and -1 strides, respectively, see Figure 5.1d), which is not otherwise discernible by looking at the global memory sequence alone. Although simple, this example shows how many simple access patterns cannot be effectively captured by using global stride patterns. More data-structures with a greater degree of interleaving is likely to cause greater aliasing in the stride tables (with limited global history), leading to poor cloning accuracy.

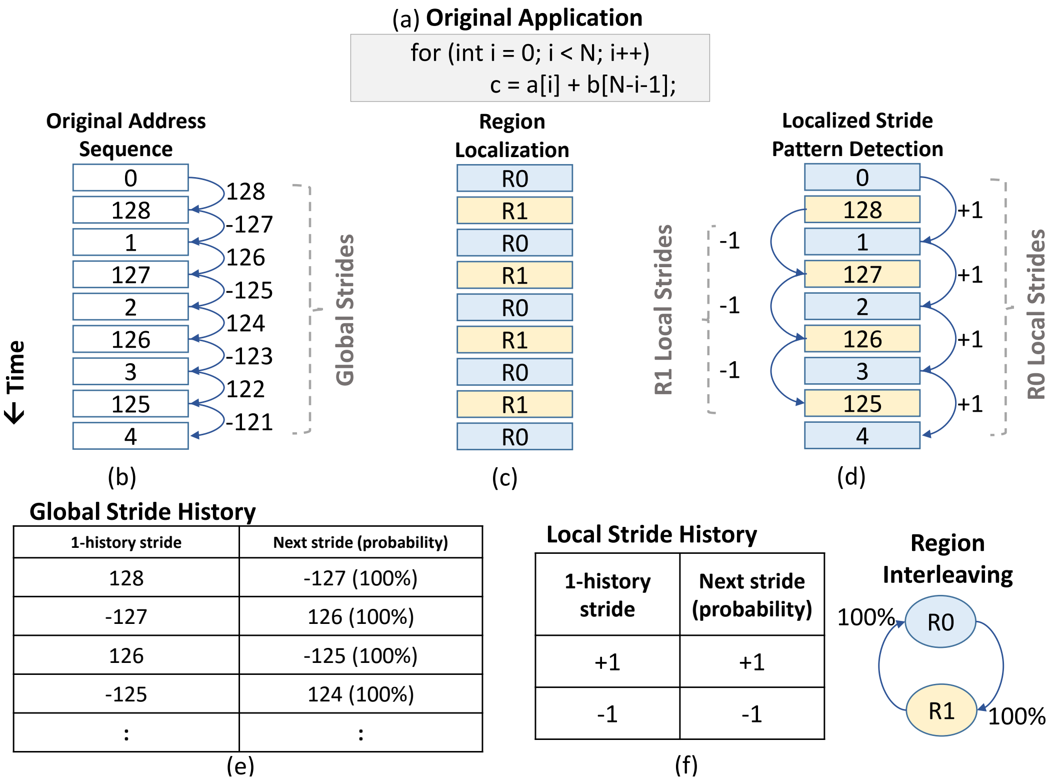


Figure 5.1: Global versus Local Memory Access Pattern Tracking.

In this dissertation, a **H**ierarchical memory **A**ccess **LO**cality modeling technique, "***HALO***", is presented that can statistically capture the spatial locality, temporal locality and memory footprint of applications, while incurring less meta-data storage overhead (which is an important consideration for portability). HALO leverages the observation that different data-structures have different locality properties and their access patterns can be detected more easily by analyzing localized access patterns. Thus, HALO discovers patterns by first decomposing memory references into localized address regions and then identifying access patterns within individual regions using repeating stride transitions. In this example, HALO localizes addresses into two regions (*R0* & *R1*) and learns stride transitions within the localized regions as shown in Figures 5.1c & 5.1d, respectively (a memory region = 64 cache-blocks). However, capturing intra-region locality patterns alone is not sufficient to recreate the original memory access behavior in the proxy benchmark. What is equally important is to capture how accesses to these individual regions are interleaved with respect to each other. HALO models the interleaving information by exploiting coarse-grained temporal locality patterns and uses it to synthesize an ordered proxy reference sequence from individual localized stream accesses (see Figure 5.1f).

5.1 HALO's Methodology

Figure 5.2 shows an overview of HALO's memory locality modeling framework. During the profiling phase ①, HALO characterizes the application's inherent memory access patterns to create a statistical workload-specific profile ②. HALO discovers memory access patterns by decomposing the original references into different regions (***"region localization"*** Ⓐ) and capturing fine-grained access patterns within individual regions using repeating stride transitions (***"intra-region stride locality"*** Ⓒ). In particular, HALO captures ***multi-level stride transition probability distributions***, which are tailored to the locality behavior of different applications, to achieve higher cloning accuracy and meta-data storage efficiency. HALO further captures how accesses to these individual localized regions are interleaved with respect to each other by tracking coarse-grained temporal locality patterns (***"inter-***

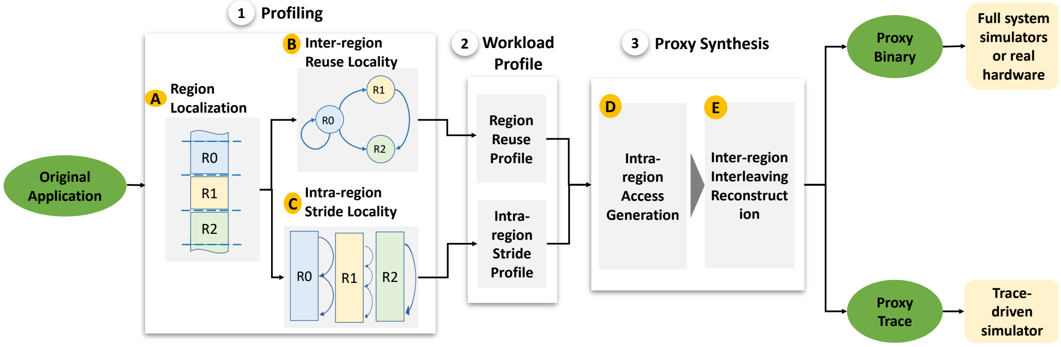

1 Profiling
Original Application
A Region Localization
R0
R1
R2
B Inter-region Reuse Locality
R0
R1
R2
C Intra-region Stride Locality
R0
R1
R2
2 Workload Profile
Region Reuse Profile
Intra-region Stride Profile
3 Proxy Synthesis
D Intra-region Access Generation
E Inter-region Interleaving Reconstruct ion
Proxy Binary
Full system simulators or real hardware
Proxy Trace
Trace-driven simulator

Figure 5.2: HALO’s Memory Locality Cloning Methodology.

region reuse locality” Ⓑ). During the proxy synthesis phase ③, HALO adopts a systematic methodology to create a miniature memory access clone of the original application based on the captured workload-specific profile that can then be used to drive cache hierarchy, TLB and memory system performance exploration. To do so, HALO first generates proxy accesses within localized memory regions by leveraging the collected intra-region stride statistics (***“intra-region access generation”*** Ⓓ) and then interleaves accesses from the localized streams using the captured reuse locality statistics (***“inter-region interleaving reconstruction”*** Ⓔ) to create an ordered proxy reference sequence.

5.1.1 Region Localization

During the *region localization* step, HALO divides the address space into fixed-size segments called ***regions*** and assigns the original memory references to different regions based on the higher-order bits of the addresses. The key idea behind region localization is that, for most applications, similar data-structures (with similar access patterns) are often laid out in continuous address segments. Accesses to such regions or data-structures often have different patterns as compared to other regions or data-structures that are accessed together. Detecting patterns within a single global access stream is usually not effective or has higher storage overhead because effects such as data-dependent control flow, program complexity, data-structure access pattern differences, data layout, etc. lead to increased entropy in the global reference patterns. In contrast, using localized pattern detection can lead to more accurate representation of access patterns. Localized pattern correlation is also leveraged by many prefetchers [49, 59, 68, 45, 105] to make prefetch predictions. HALO considers each memory region to be a contiguous 4KB segment in the memory space.

5.1.2 Intra-region Stride Locality Tracking

After localizing the original memory accesses into different regions, HALO captures fine-grained access patterns within individual regions using intra-region stride probability distributions. However, what stride history length can efficiently capture dominant intra-region stride locality behavior across different applications?

Figure 5.3 shows the cumulative fraction of intra-region stride transitions (y-axis) that can be captured using increasing history-length based stride transition tables (x-axis) without having any aliasing effects for 8 applications. It can be observed that applications have diverse locality behavior. For example, for the bwaves benchmark with highly-strided access patterns, more than 98% of the intra-region stride transitions can be summarized using a history length of 3. Similarly, both

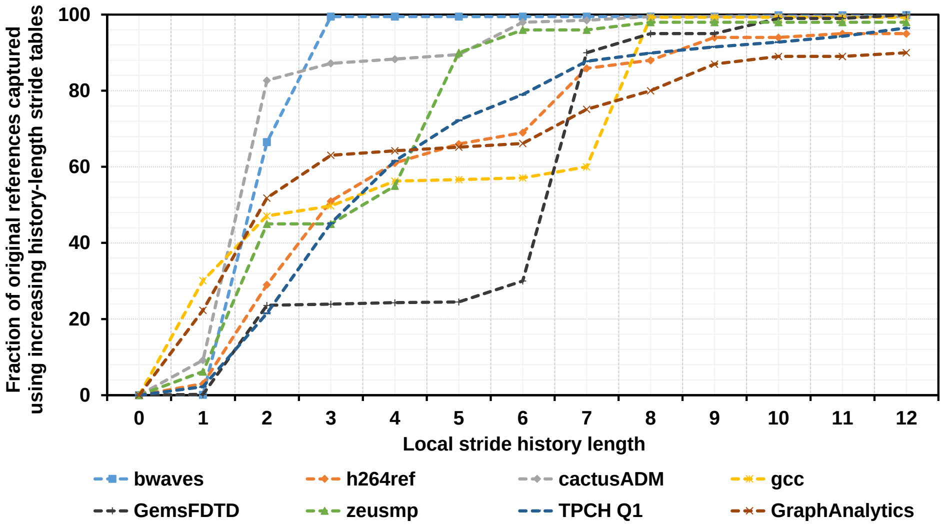
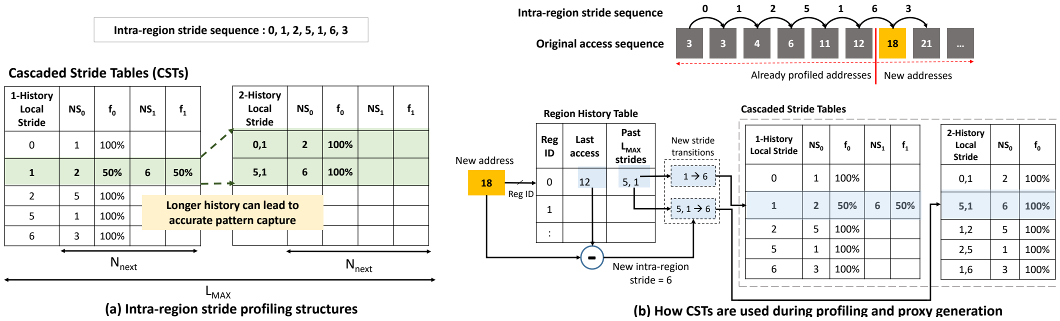


Figure 5.3: Fraction of Original Reference Patterns Captured Using Increasing History-length Based Stride Tables.

cactusADM and zeusmp benchmarks operate on a 3D array/grid and have fairly strided access patterns. However, while cactusADM iterates over the grid points in one dimension, zeusmp iterates over data points in all three dimensions. Thus, most dominant intra-region access patterns of cactusADM can be summarized using a history length of 2, but zeusmp requires slightly longer stride history length ($\sim 4-6$). On the other hand, for benchmarks such as graph analytics, which consists of many complex indirect references, using a local history length of 10 also suffers from aliasing effects in a few memory regions. Choosing a long history-length to account for locality of worst-case benchmarks would increase meta-data storage overhead for other benchmarks without providing any significant accuracy benefit, while reducing the history length would cause aliasing in the stride tables,


Intra-region stride sequence : 0, 1, 2, 5, 1, 6, 3
Cascaded Stride Tables (CSTs)
Longer history can lead to accurate pattern capture
N$_{next}$
N$_{next}$
L$_{MAX}$
Intra-region stride sequence
0 1 2 5 1 6 3
Original access sequence
3 3 4 6 11 12 18 21 ...
Already profiled addresses
New addresses
New address
18
Reg ID
Region History Table
New stride transitions
1 → 6
5, 1 → 6
New intra-region stride = 6
Cascaded Stride Tables

1-History Local Stride	NS_0	f_0	NS_1	f_1
0	1	100%		
1	2	50%	6	50%
2	5	100%		
5	1	100%		
6	3	100%		

2-History Local Stride	NS_0	f_0	NS_1	f_1
0,1	2	100%		
5,1	6	100%		

(a) Intra-region stride profiling structures

Reg ID	Last access	Past L_{MAX} strides
0	12	5, 1
1		
:		

1-History Local Stride	NS_0	f_0	NS_1	f_1
0	1	100%		
1	2	50%	6	50%
2	5	100%		
5	1	100%		
6	3	100%		

2-History Local Stride	NS_0	f_0
0,1	2	100%
5,1	6	100%
1,2	5	100%
2,5	1	100%
1,6	3	100%

(b) How CSTs are used during profiling and proxy generation

Figure 5.4: Intra-region Locality Profiling using Cascaded Stride Tables (CSTs).

leading to poor accuracy. In any case, it should be noted that the localized patterns can be captured using much shorter history lengths as compared to global memory patterns. For example, in the h264ref benchmark (see Figure 5.3), most intra-region stride transitions can be captured using a local history length of 8, while even a history length of ∼100 is not enough to capture the dominant global stride transitions [5].

In order to leverage the diverse program locality to achieve improved cloning accuracy and meta-data storage efficiency, HALO proposes tailoring the stride history length based on the application's locality needs. HALO achieves this by using a set of cascaded stride tables (CSTs) to capture the intra-region stride transitions. Each stride table tracks a longer stride history length and associates specific intra-region stride histories with the next possible strides to the same region. Figure 5.4a shows an example to demonstrate the working of the CST structures. Before moving forward, let us first clarify a few notations: CST_i is used to refer to a stride table tracking i-length stride history, L_{MAX} refers to the maximum cascading degree

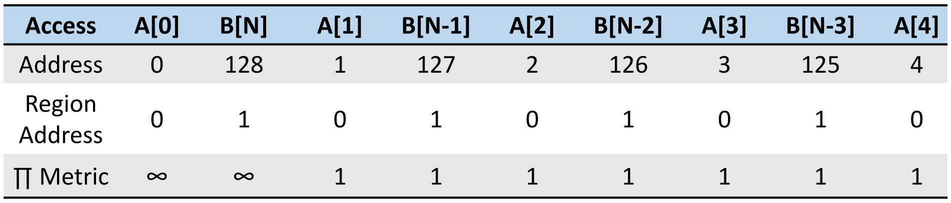
(L_{MAX} = 2 in this example), and (NS_0, f_0) refers to next-stride value and its probability of occurrence. In the original stride sequence shown in Figure 5.4a, stride $\{1\}$ is followed by strides $\{2\}$ or $\{6\}$ with equal probability, which causes aliasing in the CST_1 table. Using only 1-history transitions for proxy synthesis can lead to a different stride interleaving in the proxy versus the original application because of such pattern aliasing. The aliasing effects can, however, be eliminated in this example by capturing 2-history stride transitions in the CST_2 table. Using the CST_2 table can accurately model the stride following $\{1\}$ with 100% accuracy depending on its preceding stride ($\{0\}$ or $\{5\}$). Capturing other 2-history stride transitions in the CST_2 table (e.g., $\{2, 5\} \rightarrow \{1\}$) is not necessary as the same patterns can be captured using 1-history transitions ($\{5\} \rightarrow \{1\}$). Thus, using CSTs enables locality-specific access pattern capture; shorter history tables can efficiently capture simple/regular patterns, while more complex patterns are tracked using longer history-based stride transitions. Conceptually, using multiple cascaded tables to track histories of varying lengths is similar to the state-of-the-art TAGE branch predictor [84] or variable length delta prefetcher [88].

Figure 5.4b shows the profiling structures used for capturing the multi-level stride transitions. During a profiling interval, HALO keeps track of accesses to different regions using the *region history table* (RHT). Each RHT entry tracks the number of region accesses, past L_{MAX} intra-region strides within the region, etc. To profile a memory access, RHT is indexed using the address's region *index* and a new stride is computed based on the region's last seen address. The CST tables are updated based on the new stride and the history of last L_{MAX} strides to the region. Unfortunately, during the profiling interval, it is not known as to which history length can capture the current stride transition with least aliasing (as the entire application reference stream has not been profiled yet). Thus, during a profiling interval, HALO updates all the stride tables using the accumulated stride history of the corresponding region, where each cascaded table tracking history length L_i is updated using the accumulated last L_i intra-region strides, $\forall i \leq L_{MAX}$. For example, when address *18* (region *R0*) is profiled, both CST tables are updated using the accumulated stride history and the new stride ($\{\mathit{1}\} \rightarrow \{\mathit{6}\}$ *and* $\{\mathit{5, 1}\} \rightarrow \{\mathit{6}\}$ respectively).

At the end of the profiling interval, HALO analyzes the complexity of stride transitions captured in the CSTs (starting from the longest-history one) and identifies the minimum history length ($L^{'}_{MAX}$) that can capture the respective access patterns with least aliasing. To do so, HALO scans the CST tables (starting from the longest-history one) one-by-one and invalidates those stride pattern entries that can be captured using a shorter length history table with similar accuracy. If all the next stride probabilities in a CST_k table exceed a threshold of δ, then $L^{'}_{MAX}$ is set to be k for that application. In this example, at the end of the profiling interval, HALO post-processes the CST_2 table and invalidates the last three entries ($\{\mathit{1, 2}\} \rightarrow \{\mathit{5}\}$, $\ldots$, $\{\mathit{1, 6}\} \rightarrow \{\mathit{3}\}$) as the same patterns are captured in the CST_1 table. Also, the $\{\mathit{1}\} \rightarrow \{\mathit{2, 6}\}$ entry in the CST_1 table is invalidated as 2-history pattern needs to be captured to remove aliasing effects. Figure 5.4a shows the final state of the CST tables after post-processing. ***The final post-processed state of the CST probability distribution tables is saved for proxy generation***. During post-processing, adjacent regions with similar intra-region stride patterns can be identified and merged to form larger regions. Since individual CST tables contain a maximum of a few tens to hundreds of entries for most applications, the time overhead to manage the cascaded tables is not significant.

5.1.3 Inter-region Reuse Locality

As discussed earlier, capturing intra-region locality metrics alone is not sufficient to recreate the original memory access locality. HALO further captures how accesses to the individual regions are interleaved with respect to each other. To understand why, let us re-visit the example program in Figure 5.1. This program makes repeated accesses to the two arrays in an interleaved manner. However, during proxy generation, if the cloning framework generates accesses to the two arrays in a sequential manner (all accesses to $R0$ finish before $R1$ is accessed), the proxy program's locality will be very different from the original program. HALO captures the degree of interleaving between accesses to individual memory regions by monitoring coarse-grained temporal locality patterns using the ***region reuse distribution***



Access	A[0]	B[N]	A[1]	B[N-1]	A[2]	B[N-2]	A[3]	B[N-3]	A[4]
Address	0	128	1	127	2	126	3	125	4
Region Address	0	1	0	1	0	1	0	1	0
∏ Metric	∞	∞	1	1	1	1	1	1	1

Figure 5.5: Inter-region Reuse Locality (Π Metric) Tracking.

(Π) metric. The Π distribution captures the number of unique region accesses between successive accesses to the same region. Figure 5.5 shows an example of Π metric computation for the program discussed in Figure 5.1. The last row shows the computed Π metric (∞ represents a newly-accessed region). During proxy generation phase, the Π profile is used to reconstruct an ordered proxy memory reference sequence from individual region streams.

Modeling the Π statistics also gives HALO the ability to accurately control the memory footprint of the generated proxy (corresponding to the ∞ counts in the Π-profile). Synthesizing proxies using global stride transitions alone experiences up to 195%, 91% and 55% error in replicating the memory footprint of original benchmarks using a stride history length of 10, 30 and 60 respectively due to aliasing in the stride transition tables. Higher error in replicating the application memory footprint translates into higher TLB, cache and DRAM performance errors. On the other hand, by tracking stride transitions at a localized granularity, HALO helps to reduce the error rates by reducing aliasing effects. Nonetheless, modeling inter-region reuse locality enables HALO proxies to achieve over 99% accuracy in replicating the desired memory footprint and TLB miss rate of the original applications.

5.1.4 Proxy Generation Algorithm

Table 5.1 summarizes the key statistics that HALO captures to model the memory behavior of applications. These statistical profiles are used to generate HALO's memory locality proxies. Algorithm 2 shows HALO's proxy generation

Table 5.1: Profiled Statistics for HALO Proxy Generation

Statistic	Description
$CST = \{CST_1, \ldots, CST_{L_{MAX}}\}$	Set of cascaded stride tables with increasing history length
L_{MAX}	Maximum cascading degree
CST_i	Stride pattern table keeping stride transition counts from past *i* intra-region strides to next stride
N_{next}	Number of possible next intra-region strides
$\Pi_{RD\|Count}$	Region reuse distance histogram
RD_{MAX}	Maximum region reuse distance bin
ρ_{rw}	Fraction of write accesses
$Rate_{mem}$	Rate of memory reference generation

algorithm. The inputs to the algorithm are the statistics shown in Table 5.1. The output is the memory proxy characterized by a tuple $\{(ADDR_i, RW_i)\}$, where $ADDR_i$ refers to the i^{th} proxy address and RW_i denotes the access type. Before proxy generation, miniaturization is applied by scaling down the collected statistical input profiles by the desired scaling factor, T_{min}. Care should be exercised when choosing an appropriate scaling factor because scaling beyond a certain limit will cause inaccuracies in modeling the memory reference patterns in the proxy due to the law of large numbers. Algorithm 2 assumes the existence of a data-structure (*RegInfo*) to track the LRU history of distinct region references. The last n^{th} accessed region can be obtained by using the function *Get_Region(n)*, while the *RegInfo* data-structure can be updated as new regions get accessed using the *Update_Region()* function.

Algorithm 2 HALO's Proxy Generation Algorithm

```
 1: Input: Table 5.1 Statistical Profiles.
 2: Output: Trace $T[] = \{(ADDR_1, RW_1), \ldots, (ADDR_N, RW_N)\}$;
 3: for $i = 1, \ldots, N$ do
 4:     Sample $\pi_i$ from $\Pi$
 5:     if $\pi_i < RD_{MAX}$ then
 6:         $Reg_i$ = RegInfo.Get_Region($\pi_i$);
 7:     else
 8:         Sample $Reg_i$ uniformly in the address space;
 9:     end if
10:     ($R$, $LAST\_ADDR$, $LAST\_STR$) = RHT[$Reg_i$];
11:     $j = L_{MAX}$;
12:     while $j > 0$ do
13:         $f$ = $CST_j$.find($LAST\_STR[L_{MAX} - j : L_{MAX}]$)
14:         if $f$ == $True$ then
15:             Sample stride $S_i$ from $CST_j$;
16:             break;
17:         end if
18:         j = j - 1;
19:     end while
20:     Sample $RW_i$ from $\rho_{rw}$;
21:     $ADDR_i$ = $LAST\_ADDR$ + $S_i$;
22:     $LAST\_STR$.push_back($S_i$);
23:     $LAST\_ADDR$ = $ADDR_i$;
24:     RHT[$Reg_i$].UpdateState($LAST\_ADDR$, $LAST\_STR$);
25:     RegInfo.Update_Region($Reg_i$);
26: end for
27: return Trace[]
```

To generate the i^{th} memory address, the Π profile is sampled to obtain a region reuse distance value (line 4). The corresponding region is obtained by invoking the *Get_Region()* function. If the chosen reuse distance is greater than RD_{MAX} (corresponding to ∞ reuse distance), a new region is chosen by uniformly sampling the address space. After obtaining the region index, the *RHT* table is looked up to find the last accessed address and stride history of the chosen region. Then, the CSTs are searched one-by-one (lines 12-19), starting from the longest history-length table, by using a partial hash of the accumulated stride history. A new stride is chosen based on the longest history match in the CSTs. This ensures that the next stride assignment is done using the most accurate profiled information. Next, the i^{th} address is computed (line 21) using the last accessed address (*LAST_ADDR*) of the region and the chosen stride value (S_i). Finally, the *RHT* entry and *RegInfo* data-structure are updated based on the generated address (lines 22-25). This process is repeated until the target number of references *N* is generated.

5.1.5 Execution Phase Consideration

Most applications exhibit different locality behavior during different execution phases. For example, a program can have a large footprint or be prefetcher-friendly during a certain phase, but access a very small data segment or become prefetcher-unfriendly during other phases. Modeling such changes in program locality can lead to higher correlation between the proxy and original workloads. To account for phase behavior, HALO divides the original access sequence into fixed size intervals and tracks an independent intra-region stride and inter-region reuse profile per profiled interval period. The *RegInfo* data-structure (used for tracking region reuse) is not cleared between phases. HALO uses the per-interval stride and reuse profile information to generate a proxy sequence for the corresponding interval. The interval length is empirically chosen to be 100,000 memory references. Section 5.2.2 will discuss the sensitivity of cloning accuracy to changes in phase lengths.

5.1.6 Multi-programmed Workload Performance

When applications are co-scheduled on a CMP, memory access streams from different applications compete for the shared cache space. To model the cache-sharing behavior of co-scheduled workloads, HALO uses another statistic - the rate at which memory references are generated per application ($Rate_{mem}$). This metric accounts for the fraction of memory instructions over total instructions, instruction level parallelism and relative speed of the processor cores. As HALO proxies do not produce instruction streams other than memory references, HALO controls the distance between successive memory references based on this rate metric.

5.2 Evaluation

This section discusses the experimental setup followed by a detailed evaluation of HALO's performance cloning accuracy.

5.2.1 Experimental Setup

Evaluation is performed using 39 benchmarks from different application classes: (a) 26 SPEC CPU2006 benchmarks [93] using "ref" input set (all benchmarks except perl, sjeng and dealII due to compilation issues), (b) 6 benchmarks from the newly introduced SPEC CPU2017 [94] suite (leela, exchange2, imagick, pop2, roms and nab of SPECspeed category), (c) 3 TPC-H [100] queries (Q3, Q6, Q14) using MySQL [58] database, (d) a data serving workload (WC) based on

Yahoo! Cloud Serving Benchmark [18] framework (12GB) from Cloudsuite [26], (e) graph analytics *tunkrank* (Graph) application based on Graphlab [2] framework from Cloudsuite and *connected components* (CC) application using GraphChi [1] framework, and (f) data-caching benchmark based on Memcached [3] from Cloudsuite.

The benchmarks are profiled using a Pin-based [54] detailed simulator. The system configuration used for collecting the statistics is shown in Table 5.2. For CPU2006 benchmarks, a representative region, consisting of 250 million instructions, is identified using the simpoint [87] methodology. For the other benchmarks, a representative region is identified by fast-forwarding the benchmark execution by 10 billion instructions, and then profiling the execution of next 250 million instructions. Representative regions consisting of 250 million instructions are chosen to make the simulation runs for validation manageable. It should be noted that the HALO only uses a statistical profile as input for proxy generation, which is independent of the execution length. To extensively evaluate HALO's cloning accuracy across different cache and prefetcher configurations, a validated trace-driven cache

Table 5.2: HALO's Profiled System Configuration

Component	**Configuration**
CPU	X86_64 processor, atomic mode 4 GHz Single-core and multi-programmed runs
L1 cache	32KB, 2-way Icache; 64KB, 2-way Dcache 64B block size, LRU
L2 cache	4MB, 8-way, LRU, Shared
Main memory	4GB DDR3, 12.8 GB/sec
OS	Ubuntu 14.04

simulator is used. The simulator is validated by comparing its miss rates with the standard cache modules provided by the gem5 [13] simulator. For evaluation and testing of DRAM memory performance, the Ramulator [48] memory system simulator is used. The generated HALO proxies consist of 10-15M memory references. HALO's results are compared against the state-of-the-art WEST and STM proposals.

5.2.2 Results and Analysis

In this section, HALO's cloning effectiveness is evaluated in replicating cache, prefetcher, TLB and DRAM performance of applications over 20,000 different configurations. Two metrics are used for performance validation: error between original and proxy performance metrics and Pearson's correlation coefficient. Pearson's correlation coefficient indicates how well the proxy benchmarks track the trends in the original applications, with 1 indicating perfect correlation, and 0 indicating no correlation. During design-space exploration, computer architects consider relative performance ranking (e.g., evaluating which configuration has a lower miss rate). Considered together, the average cloning error and correlation degree shows how closely the proxy workloads perform with respect to the original workloads across different configurations.

Instructions per cycle - First, the proposed methodology is evaluated by measuring the performance of the original and proxy workloads across over $\sim 6,600$ different configurations, generated by varying the size, associativity and line-size of the L1/L2 caches and the L2 stream prefetcher. Figure 5.6 shows the

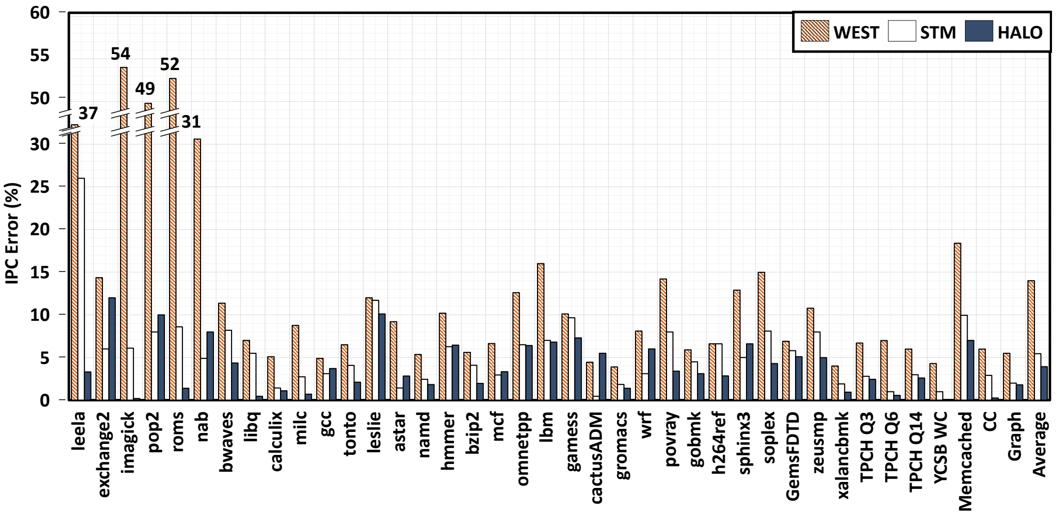


Figure 5.6: Instructions per Cycle Error of WEST, STM and HALO Proxies versus the Original Applications.

error between instructions per cycle (IPC) of the original and proxy workloads. Overall, the average error in replicating the original workload IPC for WEST, STM and HALO proxies is 14%, 5.4% and 3.9% respectively. Higher cloning accuracy of HALO proxies over both WEST and STM proxies is a result of more accurate modeling of cache, prefetcher and memory system performance. A detailed analysis of the performance implications of the individual metrics is presented in the following paragraphs. Please note that IPC is used here as a metric to validate the proposed memory model across a range of memory hierarchy configurations, but is not indicative of processor-side performance (as HALO does not model non-memory instructions).

L2 cache and prefetcher configurations - Next, HALO's effectiveness is

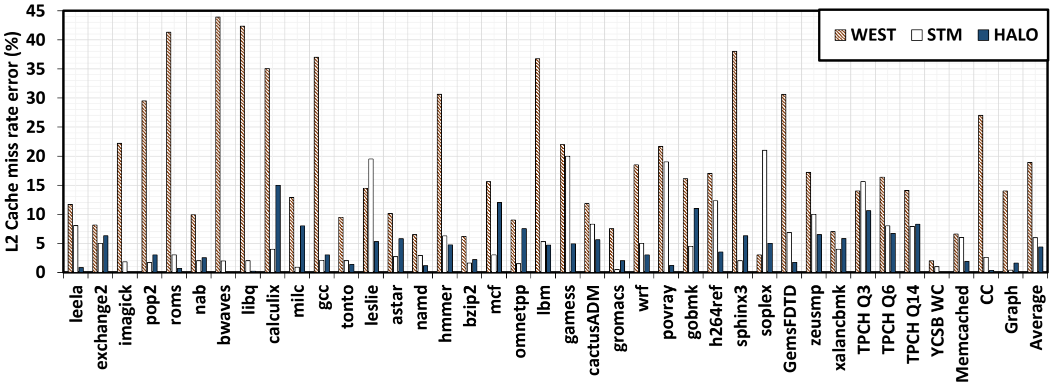


Figure 5.7: L2 Miss-rate Errors of WEST, STM & HALO Proxies across L2 Cache and Prefetcher Configurations.

tested in replicating L2 cache performance by varying the L2 cache and prefetcher configurations simultaneously. In particular, 35 different L2 cache configurations are run per benchmark by varying the cache size between 1MB-16MB, associativity between 2-32 and line size between 32/64/128. For each cache configuration, the L2 stream prefetcher configurations are also varied by changing the number of stream buffers between 8-64 and changing the prefetch degree between no-prefetching /1/2/4/8, resulting in a total of 260 different configurations per benchmark. Figure 5.7 shows the L2 miss rate error between the original and proxy benchmarks (averaged across different configurations). The average errors in replicating L2 cache miss rates for WEST, STM and HALO proxies are 18.9%, 6% and 4.4% respectively. The correlation coefficients are 77.9%, 97.5% and 98.5% for WEST, STM and HALO respectively.

As WEST does not model spatial locality, it suffers from high errors, especially when prefetchers are enabled for prefetch-friendly benchmarks (e.g., bwaves, libquantum). Also, WEST captures stack distance distributions at a cacheline granularity, and thus, suffers from high cloning errors when cache line-size changes, cache size increases, etc. Because it models spatial locality patterns, STM performs better than WEST. However, for many benchmarks, like leela, h264ref, exchange2, povray, STM experiences high aliasing in its global stride transition tables, leading to poor cloning accuracy of STM proxies. Also, STM captures global stride transitions at a cacheline granularity. Thus, STM proxies do not capture spatial locality within cachelines and perform poorly when cacheline size is varied for some benchmarks (e.g., zeusmp).

Overall, HALO outperforms both WEST and STM. HALO performs well even for benchmarks, like leela, h264ref, povray by using a local history depth of 8; the dominant access patterns of these benchmarks cannot be captured by STM even when using a global stride history depth of 80. This clearly shows the effectiveness of HALO's localized pattern detection methodology. By leveraging multi-granularity stride transitions, HALO not only performs well for benchmarks like libquantum, which have regular strided patterns, but also for benchmarks like gcc and sphinx3, which make a lot of irregular data-structure accesses, or bzip2, which has a significant fraction of control-flow dependent loads. HALO experiences high cloning errors with calculix and gobmk benchmarks (14.5% and 11% respectively), but the high L2 miss rate errors occurs systematically for configurations with very few L1 cache misses; the average L2 MPKI error is $\leq$0.01, which causes insignificant impact on IPC ($\leq$ 1% and 3% for calculix and gobmk respectively) as shown

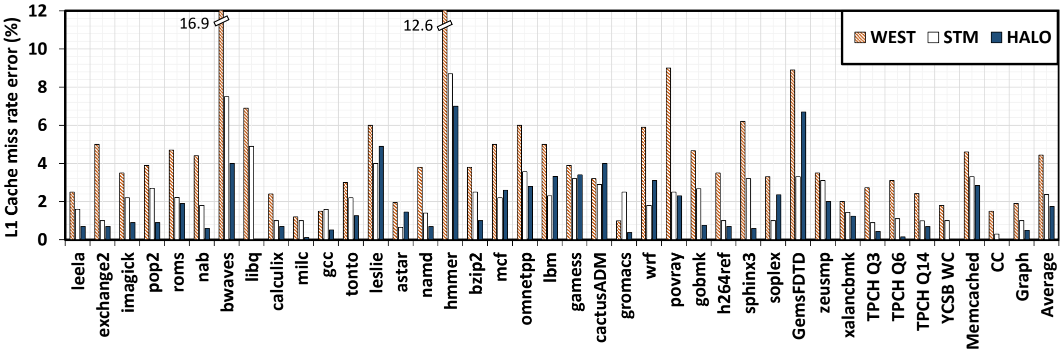


Figure 5.8: L1 Miss-rate Errors of WEST, STM and HALO Proxies across L1 Cache and Prefetcher Configurations.

in Figure 5.6. With a stride history depth of 8, benchmarks like TPC-H Q3 (complex *join* operation across three database tables) and mcf (operates on an array of pointers data-structure) experience ∼10-11% error with a stride history depth of 8 due to aliasing; increasing local history depth to 14 reduces errors to ∼5.8%, but increases profile sizes.

L1 cache and prefetcher configurations - Next, HALO's L1 performance cloning accuracy is evaluated across 40 different L1 cache configurations per benchmark (varying the cache size from 4KB-128KB, associativity from 2-16 and cache line-size between 32B-128B). For each cache configuration, the L1 stream prefetcher configurations are also changed by modifying the stream detection window between 8/16/32 and the prefetch degree between no-prefetching/1/2/4/8, resulting in 264 configurations per benchmark. Results showing the L1 cache miss rate errors are shown in Figure 5.8. Overall, the average L1 cache miss rate error between original and proxy workloads is 4.5%, 2.4% and 1.8% for WEST, STM and HALO respectively.

STM captures temporal locality using per-set LRU stack distance distributions for a 16 KB, 2-way L1 cache, however it does not track any statistics related to access distribution or ordering across cache sets. As a result, for benchmarks such as bwaves, libquantum, STM proxies produce different conflict behavior across cache sets when L1 test configurations differ from the baseline configuration, resulting in higher cloning errors. Owing to LRU-stack based modeling of temporal locality behavior, WEST experiences higher errors when test configurations (especially, cacheline size) deviate from the baseline configuration. WEST's performance degrades further when prefetching is enabled due to not modeling spatial locality behavior. Overall, HALO outperforms both WEST and STM by exploiting higher predictability in localized memory access streams even with shorter history lengths. HALO experiences higher L1 performance modeling error for hmmer and GemsFDTD benchmarks (∼7%) as HALO does not model inter-region spatial locality, which leads to cloning inaccuracies, especially with prefetching.

TLB and page size configurations - The next set of experiments test the TLB performance cloning accuracy of WEST, STM and HALO proxies (see Figure 5.9). For these experiment,s the number of TLB entries is varied between 8-128 and page size between 1KB-16KB (total 25 configurations per benchmark). Overall, WEST, STM and HALO have 9.2%, 2.4% and 0.7% error in replicating TLB miss rates of original applications. WEST generates a random memory address for any references that miss in the L2 cache. This causes higher deviation in the memory footprint and fraction of active pages (during any interval) between WEST proxies

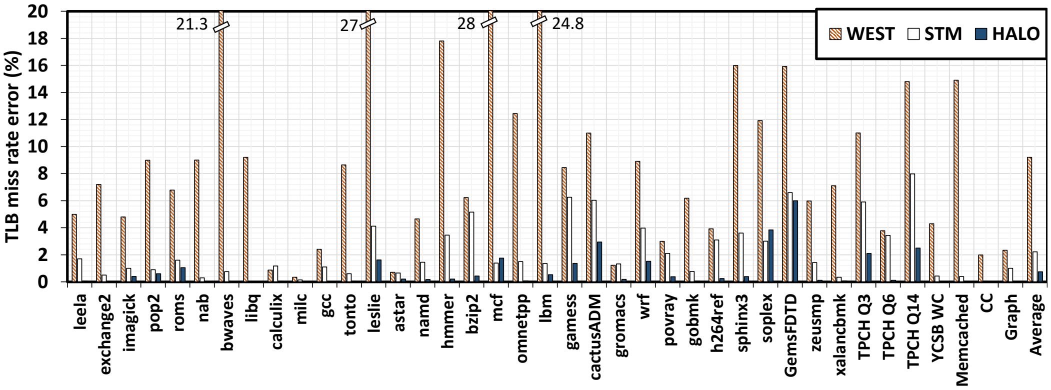


Figure 5.9: TLB Miss-rate Errors of WEST, STM and HALO Proxies across Different TLB & Page-size Configurations.

and original applications. STM is more accurate in replicating TLB behavior than WEST; however, aliasing in STM's global stride tables also causes the memory footprint and TLB performance of STM proxies to deviate from original applications. In contrast, by leveraging coarse-grained reuse locality to model inter-region interleaving, HALO can accurately model TLB performance across most benchmarks except GemsFDTD. HALO proxies are generated using a base region size of 4KB. In GemsFDTD, increasing the page size affects the inter-region access interleaving order, which results in higher TLB errors. For most other benchmarks, changing the TLB or page size configuration has minimal impact on HALO's accuracy. Overall, HALO outperforms both WEST and STM, achieving an average accuracy of 99.3%.

Phase-level cache performance modeling - As discussed previously, HALO models memory access locality at a phase granularity to accurately capture fine-

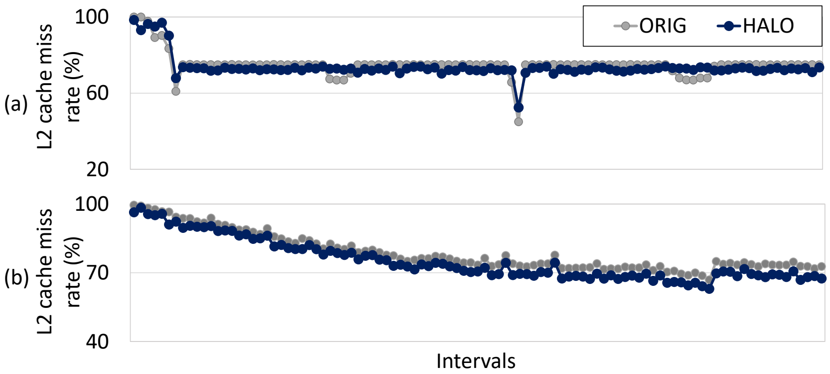


Figure 5.10: Example Showing Phase-level Cache Performance Modeling for (a) GemsFDTD and (b) Graph Analytics

grained locality changes. Figure 5.10 compares the phase-wise L2 cache miss rates of the original and HALO workloads for the Graph-analytics and GemsFDTD benchmarks. Every corresponding phase of the original and proxy workload is aligned after accounting for miniaturization. It can be observed that the cache miss rate of Graph analytics workload varies between ∼70-100% between the different phases and the HALO proxy follows the original application's trends very closely with an average error of 1.6%. Similarly, although GemsFDTD experiences slightly higher average error, the HALO proxy still captures the relative trends across different execution phases quite accurately. HALO has similar phase-level cloning accuracy across other benchmarks as well.

DRAM Performance - The next set of experiments validate the effectiveness of HALO proxies to be used for design exploration of memory subsystem in lieu of the original workloads. For this study, the Ramulator [48] memory system simulator is used in combination with the cache simulation model. Over 25 different

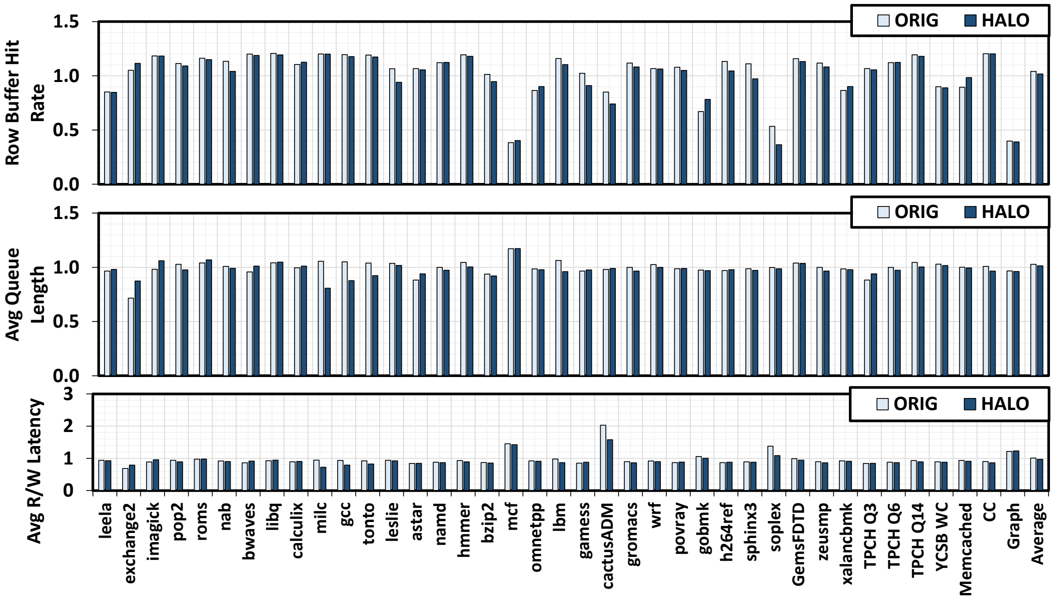


Figure 5.11: Comparing DRAM Performance of HALO and Original Applications across Different DRAM Configurations.

DRAM configurations per benchmark are run by changing the DRAM bus width (4-16 bytes), bus frequency (800MHz-1GHz) and DRAM address mapping schemes (RoBaRaCoCh/ChRaBaRoCo) by swizzling the address decoding bits, while simultaneously varying the L2 cache size and associativity. The original and proxy workloads are compared across three key memory system performance metrics: DRAM row buffer hit rate, average memory controller queue size and average read/write latency (see Figure 5.11). By accurately capturing the spatial and temporal locality of applications, HALO proxies perform closely with respect to the original application, achieving an average error rate of 2.3%, 0.7 and 4% for DRAM row buffer hit rate, average queue length and average read/write latency respectively.

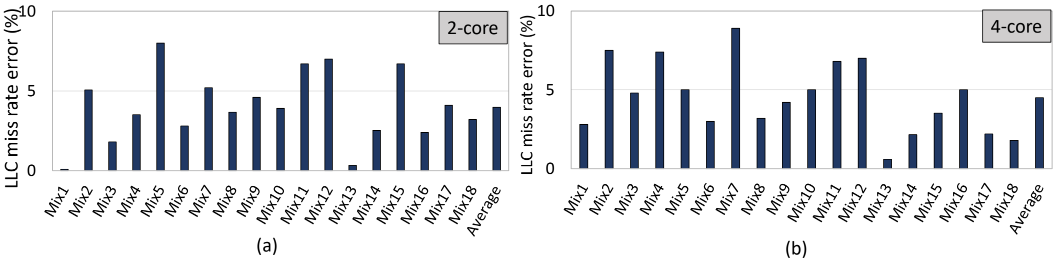


Figure 5.12: Multi-programmed Performance Error of HALO Proxies for (a) 2-core and (b) 4-core Workload Mixes

Multi-programmed workloads - The next set of experiments show how accurately HALO proxies, which are generated for benchmarks running in stand-alone mode, can replicate shared cache behavior when co-scheduled with other applications. Applications are first categorized according to their L2 miss rates and then, 18 different benchmark mixes are randomly chosen to be co-scheduled in a 2-core and 4-core setup. 40 different shared L2 cache configurations are evaluated per mix by varying the cache-size, associativity, line-size and replacement policy of the last-level cache. Overall, the average error in shared L2 cache miss rate between HALO and the original multi-programmed workloads is 4% and 4.9% for 2-core and 4-core configurations, respectively (see Figure 5.12).

Meta-data overhead - Meta-data overhead impacts portability of the proxy generation process. Meta-data profiles can be on the order of tera-bytes, if not captured efficiently, and proxy synthesis can be difficult when large profiles have to be processed. The profiles need to saved and transferred to designers in order to replay and synthesize proxies. Figure 5.13 compares the meta-data overhead

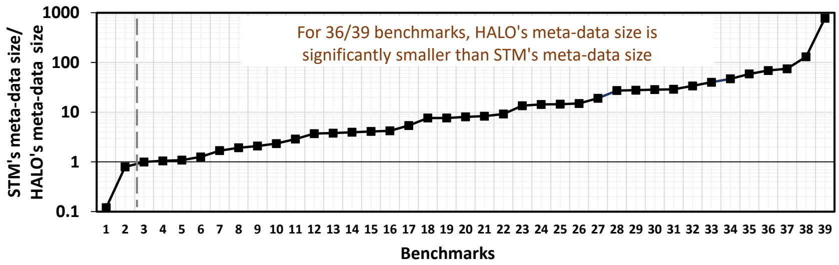


Figure 5.13: Meta-data Size of HALO versus STM (Note Y-axis in Log-scale).

of STM and HALO proxies across the 39 benchmarks. It can be observed that, by exploiting higher predictability in localized memory access streams combined with an application-locality-specific multi-level stride capture mechanism, HALO achieves, on average, a ∼39X reduction in meta-data storage size as compared to STM, while also outperforming STM across all the evaluated performance metrics. HALO's meta-data is also up to 29X smaller than gzip-compressed trace sizes. WEST does not capture spatial locality; as a result, it suffers from significantly higher cloning errors. Also, WEST's statistics are directly proportional to the profiled cache configuration, and as a result the size overhead of WEST's statistics becomes significantly higher for larger caches (e.g., meta-data overhead exceeds 2.5GBs per application for modeling a modern-day 16GB DRAM cache).

5.2.3 Sensitivity Studies

In this section, the sensitivity of HALO's performance to changes in several different parameters is studied by measuring the correlation between the proxy and original workloads across different L1 cache and prefetcher configurations. First, HALO's sensitivity to the region size is evaluated by varying it from 1KB - 16KB (see Figure 5.14a)). The *Dyn* data-point corresponds to choosing the best region ize for each application statically. As region size increases, average correlation slightly drops because of higher entropy in larger region patterns, which is difficult to capture using the same history depth without increasing aliasing. Smaller region sizes lead to accurate intra-region pattern capture, but reducing the region size below 1KB resulted in reduced performance correlation, especially with prefetching, because of not modeling inter-region spatial locality. Figure 5.14b shows HALO's performance sensitivity to the profiling interval size. As the interval size reduces, correlation improves because of accurate capture of phase-level performance patterns. However, having a very small profiling interval increases the profile size correspondingly. It can be observed that a profiling interval of 100,000 memory references provided the best balance of accuracy and meta-data overhead. Next, the impact of trace miniaturization factor on cloning accuracy is evaluated (see Figure 5.14c). As HALO relies on statistical convergence to generate the proxies, the scaling factor depends on the original number of accesses because of the law of large numbers. It can be observed that the performance correlation holds well with 10 million memory references.

5.3 Summary

This chapter presented a novel memory locality modeling framework, HALO, that accurately models the spatial and temporal locality of applications to create miniature memory access proxies. HALO isolates global memory references

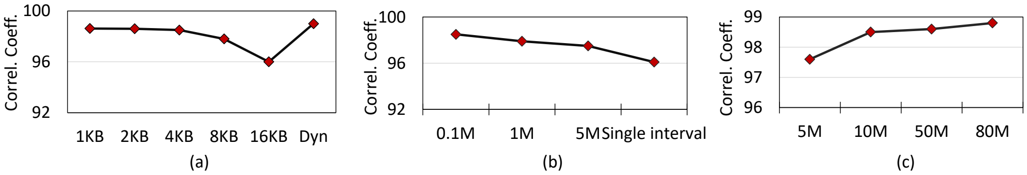


Figure 5.14: Impact of Changing the (a) Region Size, (b) Profiling Interval Period, (c) Trace Length on Profiling Accuracy.

into individual localized regions and captures intra-region access patterns using fine-grained spatial locality patterns. To achieve greater meta-data storage efficiency, HALO captures multi-level stride patterns tailored to the application's locality behavior. HALO synthesizes memory access streams from individual, localized stream accesses by modeling the degree of interleaving between accesses to different regions using coarse-grained temporal locality metrics. HALO achieves over 98.3%, 95.6%, 99.3% and 96% accuracy in replicating the performance of prefetcher-enabled L1 & L2 caches, TLB and DRAM performance, respectively across over 20,000 different cache, prefetcher, TLB, page-size and DRAM configurations. HALO also outperforms the state-of-the-art workload cloning proposal in terms of cloning accuracy across all the evaluated metrics, while using ∼39X less meta-data storage.

Chapter 6

G-MAP: Statistical Pattern Based Modeling of GPU Memory Access Streams

In the past decade, GPUs have emerged as a popular computation platform for applications beyond graphics. Programmers exploit these massively parallel architectures in a variety of domains (e.g., linear algebra and bio-informatics). GPUs leverage large amounts of parallel hardware combined with light-weight context switching among thousands of threads to hide the impact of long memory latencies and improve performance. However, many recent studies [35, 52] have shown that GPU performance is still limited by the long off-chip memory latencies. Hence, on-chip caches have been adopted in mainstream GPUs [63] to reduce the memory access latency and the off-chip memory traffic. However, GPU cache performance is often sub-optimal due to limited per-thread cache capacity, limited number of MSHRs, etc. Thus, optimizing the performance of GPU applications requires evaluating new memory hierarchy designs.

Early design space exploration of GPUs is traditionally done using detailed cycle-accurate simulators [6, 80]. Although accurate, simulators are often very slow, which severely limits the efficiency of extensive design-space exploration [109]. Recently, a few researchers have also proposed analytical models [61, 97] to estimate GPU cache performance. Although analytical models are fast, their scope is often limited (model limited degree of parallelism [61], applicable for L1 caches [61, 97], etc.). Furthermore, effective modeling techniques require access to either the application source code or memory traces, which are often inaccessible due to their proprietary nature. While several memory locality modeling techniques have been proposed for CPU applications [5, 7, 42, 11] to address such challenges, no such suitable solutions exist for cloning GPU memory access patterns.

To bridge this gap, this chapter proposes **G-MAP** [76], a novel methodology and framework that statistically models the regularity in code-localized memory access patterns of GPU applications and models the parallelism in GPU's execution model to create miniaturized **G**PU **M**emory **A**ccess **P**roxies. G-MAP proxies closely mimic the performance of original applications and enable extensive GPU memory system design space exploration. The following sections first present a brief background on the baseline GPU architecture and then a detailed description of G-MAP's methodology.

6.1 GPU Background

GPUs consist of a collection of data-parallel SIMD cores (streaming multiprocessors (SMs) in NVIDIA GPUs or compute units in AMD GPUs) as shown in Figure 6.1a. Each SM fetches and decodes a group of threads (warps in NVIDIA GPUs or wavefronts in AMD GPUs) then executes them in lockstep, following a single instruction multiple thread (SIMT) model. GPUs support multiple types of on-chip caches to utilize memory bandwidth efficiently. Each SM has private L1

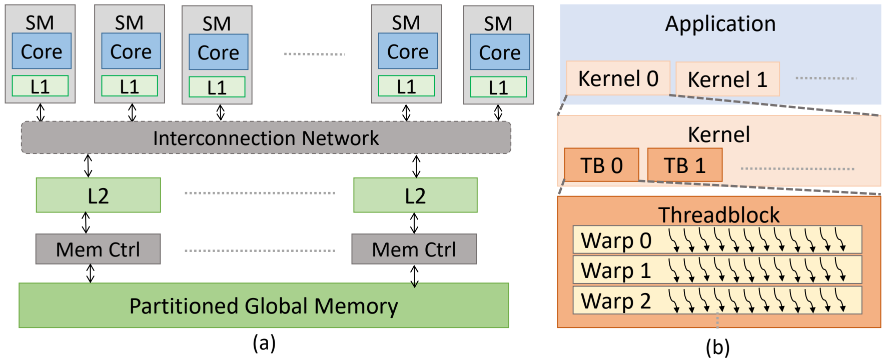


Figure 6.1: (a) GPGPU Architecture (b) GPU Application Model

data cache, texture cache, constant cache and shared memory. Global memory is partitioned and all SMs are connected to the memory modules by an interconnection network. Each memory controller consists of a shared L2 cache slice and the DRAM partition.

Figure 6.1b shows the GPU software execution model. A GPU application consists of several kernels. Each kernel is comprised of a grid of scalar threads and each thread has a unique identifier which is used to divide up work among the threads. Within a grid, threads are split into groups of threads called threadblocks (TB) or concurrent thread arrays (CTA). Threads are distributed to SMs at the granularity of entire threadblocks and multiple threadblocks can be assigned to a SM (if resources permit). Threads in a threadblock are further sub-grouped into warps (a warp is the smallest execution unit sharing the same program counter). In our baseline system, a warp consists of 32 threads. For memory instructions, a memory request can be generated by each thread and up to 32 requests are merged when these requests access the same cache line(s). Thus, only one or two memory requests are generated per warp if requests in a warp are highly coalesced.

6.2 G-MAP's Methodology

Figure 6.2 shows an overview of G-MAP's proxy generation framework. During the profiling phase ①, G-MAP characterizes the GPU application's inherent locality and parallelism patterns (e.g., thread hierarchy, spatial locality and temporal locality) to create a workload-specific statistical profile ②. Details of the different profiles captured by G-MAP will be discussed later in this section. During the clone generation and modeling phase ③, G-MAP adopts a systematic methodology to create a locality- and parallelism-aware clone of the application based on

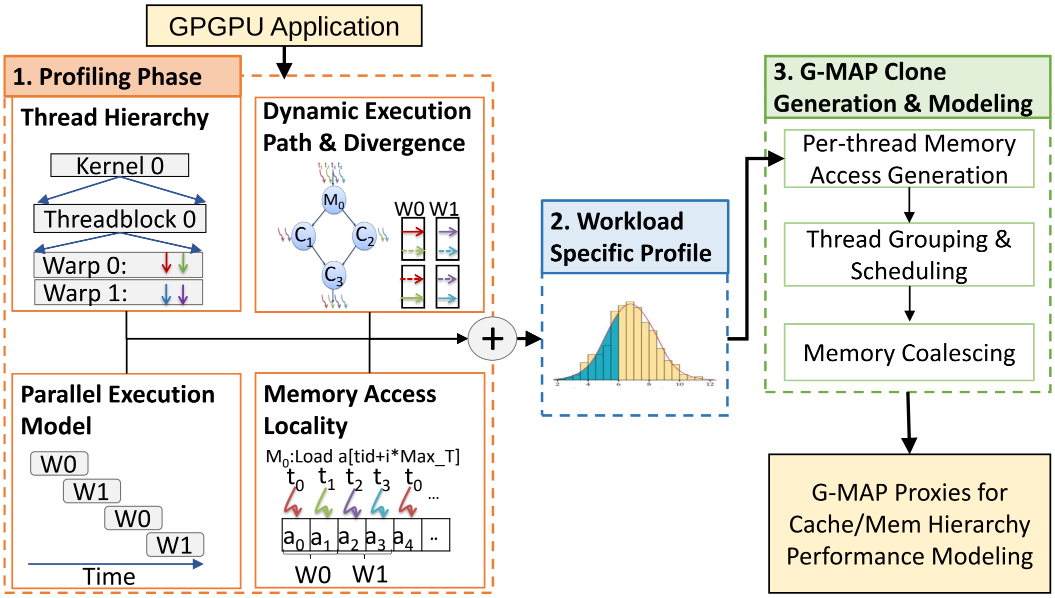

GPGPU Application
1. Profiling Phase
Thread Hierarchy
Kernel 0
Threadblock 0
Warp 0:
Warp 1:
Dynamic Execution Path & Divergence
M_0
C_1
C_2
C_3
W0 W1
Parallel Execution Model
W0
W1
W0
W1
Time
Memory Access Locality
M_0:Load a[tid+i*Max_T]
t_0 t_1 t_2 t_3 t_0 ...
a_0 a_1 a_2 a_3 a_4 ..
W0 W1
+
2. Workload Specific Profile
3. G-MAP Clone Generation & Modeling
Per-thread Memory Access Generation
Thread Grouping & Scheduling
Memory Coalescing
G-MAP Proxies for Cache/Mem Hierarchy Performance Modeling

Figure 6.2: G-MAP's Framework

the workload-specific profile, which can be used to drive GPU cache & memory performance exploration.

G-MAP exploits three key features of GPU execution to model memory access behavior using a set of statistical profiles. First, although GPU's execution model supports running thousands of threads, we observed that the ***dynamic memory execution paths*** executed by most threads can be summarized using a small set of dominant profiles. Second, most GPGPU memory operations access memory locations by exploiting a linear transformation based on the *index* (*tid*) of the thread accessing memory. This leads to high degree of regularity in how consecutive threads access different memory locations (***inter-thread locality***) for the same instruction and how individual threads access memory locations during successive iterations of the same instructions (***intra-thread locality***). G-MAP exploits this predictability in both inter- and intra-thread locality to create a memory access trace per thread, ordered based on the thread's dynamic memory execution profile. Third, synthesizing ordered per-thread memory traces alone (without accounting for GPU's parallel execution model) is not sufficient to replicate the cache/memory performance. To account for the parallelism, G-MAP leverages ***per-core warp queues*** and a ***coordinated scheduling policy*** to generate ordered per-core memory access sequences from the set of ordered per-thread accesses.

G-MAP maintains the same grid and TB dimensions as the original application. It follows Fermi's [63] execution model to group threads into threadblocks & warps based on section G.1 of CUDA programming guide [62]. G-MAP also implements a memory coalescing model to combine memory requests based on

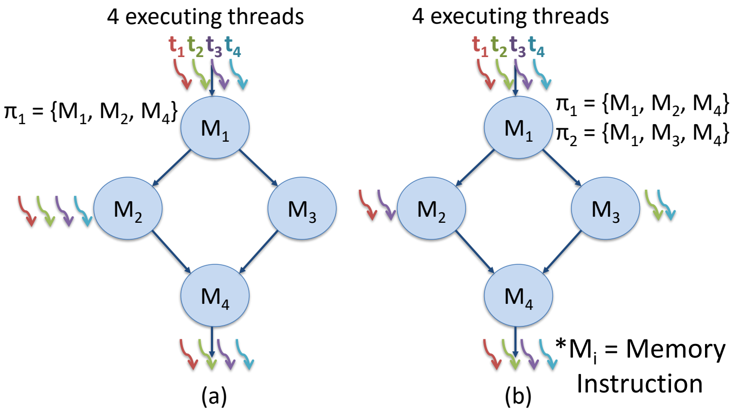


Figure 6.3: Dynamic Memory Execution Profile Capture (a) Without and (b) With Control-flow Divergence

section G.4.2 of CUDA programming guide [62]. Coalescing is modeled before applying the memory locality analysis, as it significantly reduces the computational and memory complexity of the G-MAP model. In the following sections, the profiles collected by G-MAP are first described, followed by the performance cloning algorithm.

6.2.1 Dynamic Memory Execution Profile

A GPU kernel typically executes thousands of threads. Owing to the CUDA or OpenCL execution model, every thread within a kernel executes the same sequence of instructions (computation & memory) in the absence of control path divergence. G-MAP leverages this observation to capture a single dynamic memory instruction profile (denoted as the π profile) for a *base* thread, as a representation of the sequence of dynamic memory instructions executed by all threads. For example,

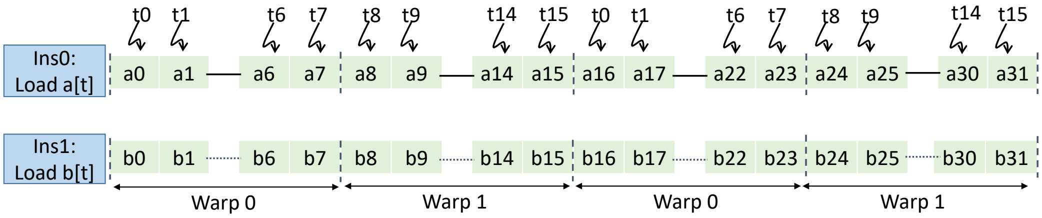


Figure 6.4: Example Showing Intra-thread and Inter-thread Strides with Two Warps Adding Elements of Two Arrays

in Figure 6.3a, all 4 threads follow the same path leading to a single dominant π profile. Of course, this assumption is valid only in the absence of control-flow related divergence effects, which can cause individual threads to execute different paths. Section 6.2.4 discusses how G-MAP accounts for such diverging effects. Nevertheless, the CUDA programming guide recommends writing programs with minimal control-flow divergence, as divergence negatively impacts warp occupancies and performance. The π profile is used for synthesizing an *"ordered"* per-thread proxy memory address sequence. G-MAP also exploits code-localization (for every static instruction in the π profile) to capture memory access patterns, as we will discuss next.

6.2.2 Inter-thread Memory Access Locality

As work distribution in a kernel is primarily done using the *tid* in most GPU applications, GPU memory operations are often a linear function of the *tid* of the thread accessing memory. Since adjacent threads differ by an index of 1, offset between addresses accessed by adjacent threads is often fixed. For example, Figure 6.4 shows such a kernel with two warps adding two arrays (a, b) under the SIMT model. Here, each warp is composed of 8 threads. We can observe how the consecutive threads access different elements of the two arrays in a regular manner with an inter-thread stride of 1.

Table 6.1 shows the dominant memory instructions, their frequency, the most dominant PC-localized inter-warp stride (after coalescing requests from threads within each warp) and stride frequency (columns 2-5) across 10 GPGPU applications (benchmark details are provided later). We can observe that, across most applications, there exists significant inter-thread memory access regularity for the dominant instructions. G-MAP captures this synergy in memory access patterns across threads in the form of a per-static instruction, inter-thread stride distribution. Later, during proxy generation, G-MAP exploits this information to generate the base addresses of every static instruction executed by each thread, starting from an initial estimate of the base addresses accessed by the *base* thread. Choice of the initial base addresses can help to create obfuscated proxy memory access sequences for proprietariness.

6.2.3 Intra-thread Memory Access Locality

Most GPU applications also exhibit regularity in how individual threads access different memory locations during successive iterations of the same instructions (e.g., in a loop). Considering the same example in Figure 6.4, using its unique *tid*, each thread accesses some elements of the two arrays (e.g., t_0 accesses the 0^{th}, 16^{th} and so on array elements, and other threads follow a similar trend). In all, a

Table 6.1: GPGPU Application Memory Access Patterns

Application	Mem PC	%Mem Freq	Inter-warp		Intra-Warp	
			Dom. Stride	%Stride	Dom. Stride	Reuse
Heartwall	0x900	81%	128	51.9%	64	High
	0x4a0	5%	128	51.9%	-128	
	0x4a8	3.8%	128	51.9%	1024	
BP	0x3F8	19.4%	128	75%	128	Med
	0x408	19.4%	128	64.1%	-128	
	0x478	19.4%	128	67.1%	128	
kmeans	0xe8	∼100%	4352	78.2%	-128	High
SRAD	0x250	31.2%	16384	78%	-8192	Low
	0x230	31.2%	16384	75%	-8192	
	0x350	31.2%	16384	80%	-8192	
SP	0xd8	48%	128	88%	4096	Low
	0xe0	48%	128	88%	4096	
CP	0x208	25%	2048	78.2%	-1024	Med
	0x218	25%	2048	78.2%	-1024	
	0x220	25%	2048	78.2%	-1024	
BLK	0xF0	20%	128	77.6%	245760	Low
	0xF8	20%	128	77.6%	245760	
	0x100	20%	128	77.6%	245760	
LUL	0x1c85	4%	352	26%	-128	Low
	0x1ca8	4%	352	26%	-128	
	0x1cc8	4%	352	26%	-128	
LIB	0x1c68	46%	128	57%	19200	High
	0x1ce0	46%	128	57%	19200	
	0x1b40	4%	128	57%	19200	
FWT	0x458	12%	128	88.6%	–	Med
	0x460	12%	128	88.6%	19200	
	0x478	12%	128	88.6%	19200	

thread with tid m accesses $m + (j * \text{Total\_Threads})$ elements of an array (where j represents the currently processed section of data) with an intra-thread stride of 16.

G-MAP exploits this regularity in intra-thread memory access patterns to clone the dynamic memory trace of each thread (memory access ordering is based

Memory Access	X[0]	X[1]	X[2]	X[3]	X[1]	X[2]	X[3]	X[0]
Cacheline	0	0	1	1	0	1	1	0
Reuse distance	∞	0	∞	0	1	1	0	1

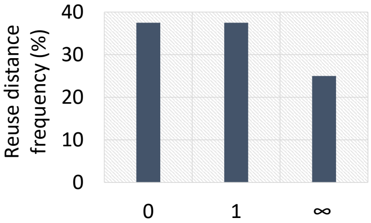


Figure 6.5: Reuse Distance Computation Example

on the π profile). G-MAP specifically leverages two key intra-thread locality metrics: (a) PC-localized stride distribution and (b) reuse distance. G-MAP captures the distribution of dominant intra-thread strides per PC. Reuse distance is an effective model of temporal locality [55, 61, 97]. It is defined as the number of distinct data elements accessed between the current and the previous access to the same data element. G-MAP tracks intra-thread reuse in the form of LRU stack distance distribution [55] (see Figure 6.5 for a reuse distance computation example). Table 6.1 shows the most dominant PC-localized intra-thread stride (after coalescing) and reuse frequency (low, medium, and high reuse implies $\leq$30%, 30 - 70% and $\geq$70% reuse, respectively) across a set of GPU applications (columns 6-7). To synthesize the per-thread proxy sequence, G-MAP generates a memory address for each dynamic memory instruction by first trying to satisfy any dominant intra-thread reuse distance (sampled from the reuse histogram) using an appropriate intra-thread stride value (if possible), followed by sampling a stride value from the intra-thread stride histogram.

6.2.4 Control-flow Divergence

So far, the discussion has assumed that all threads within a kernel execute the same sequence of memory operations, which is represented as the π profile. Even in the presence of control flow divergence, it is observed that for most applications, the dynamic memory execution profiles of individual threads can still be summarized using a small set of dominant profiles and their corresponding frequencies (see Figure 6.3b for an example kernel with two unique π profiles). To do so, G-MAP clusters the dynamic memory instruction profiles based upon their inherent similarity. For a given pair of memory instruction profiles π_i and π_j, their similarity is defined as the total number of identical entries in sequence. Two profiles belong to the same cluster if their similarity is above a certain threshold, Th (Th is empirically chosen as 0.9 in this study).

6.2.5 Scheduling Policy

Prior research has shown that the order of execution of threads (a.k.a scheduling policy) affects memory hierarchy performance. G-MAP follows Fermi's execution model to determine how threads execute together on a single core. G-MAP assigns threadblocks to cores in a round-robin (RR) fashion until they are full; new TBs get scheduled when the running TBs finish execution. Threads within each TB are sub-grouped into warps and threads within a warp are scheduled simultaneously. To account for GPU's parallel execution model, G-MAP leverages the idea of a per-core warp queue. Initially, the queue is filled with all active warps (from one or more TBs) ordered by the warp identifier (*tid / warp size*). In the simplest form, so long as the queue is not empty, a warp is selected based on RR policy and a single memory request is processed per thread. As a warp finishes a memory request, it is delayed in proportion to the request's latency. This is equivalent to the popular loose round robin (LRR) warp scheduling policy adopted in GPUs. Since G-MAP does not model the detailed GPU core, it captures the effect of other scheduling policies using a simple metric, $SchedP_{self}$, which is defined as the probability of scheduling the same warp consecutively. Although approximate, it can estimate cache & memory performance across different scheduling policies with sufficient accuracy. G-MAP models TB-level synchronization by capturing synchronization information in the π profiles and using that information to control the scheduling policy (if needed).

6.2.6 Proxy Generation and Modeling

This section will discuss how G-MAP leverages the measured statistical features to generate memory clones for evaluating GPU memory hierarchy performance. Formally, the collected features can be characterized by a 5-tuple (Π, Q, B, P_S, P_R). $\Pi = \{\pi_1, \pi_2, \ldots, \pi_M\}$ denotes the set of M dominant dynamic memory instruction profiles. Q is a probability measure on Π. $B = \{b(1), b(2), \ldots, b(N)\}$ denotes the base addresses of all N static instructions corresponding to the π profiles.

$$P_S = \{(P_E^{(1)}, P_A^{(1)}), \ldots, (P_E^{(N)}, P_A^{(N)})\}$$

contains a set of distributions $(P_E^{(i)}, P_A^{(i)})$ for each unique static instruction i. Here $P_E^{(i)}$ and $P_A^{(i)}$ denote the distribution of inter-thread stride and intra-thread stride

Algorithm 3 Trace Generation for Thread t

```
1:  Input: $\pi_i, B, P_S, P_R^{(i)}$;
2:  Output: $T_t[]$: Memory access for each instruction in $\pi_i$
3:  Initialize: B' = B;
4:  for $j^{th}$ instruction in $\pi_i$ do
5:      $k = \pi_i[j]$;
6:      if instruction $k$ is being generated for the first time then
7:          Sample offset from inter-thread stride distribution $P_E^{(k)}$;
8:          $T_t[j] = b(k)$ + offset;
9:          $b(k) = b'(k) = T_t[j]$;
10:     else
11:         Sample reuse from reuse distance distribution $P_R^{(i)}$;
12:         if $T_t[j-1-reuse] - T_t[j-1] \in supp(P_A^{(k)})$ then
13:             $T_t[j] = T_t[j-1-reuse]$;
14:         else
15:             Sample stride from intra-thread stride distribution $P_A^{(k)}$;
16:             $T_t[j] = b'(k) + stride$;
17:             $b'(k) = T_t[j]$;
18:         end if
19:     end if
20: end for
21: return $T_t[]$
```

histograms, respectively. Finally, $P_R = \{P_R^{(1)}, \dots, P_R^{(M)}\}$ denotes the collection of reuse distance distribution for each dominant memory instruction profile π.

Algorithm 4 describes G-MAP's proxy generation steps. First, G-MAP assigns a π profile to each executing thread (line 5). Next, G-MAP generates a trace for each executing thread, which is ordered based on the memory execution sequence provided in the π profile (Algorithm 3). To generate the per-thread memory trace, G-MAP uses the inter-thread stride distribution to assign base addresses for the first execution instances of every memory instruction executed by the thread (lines 6-10, Algorithm 3). For successive dynamic executions of the memory in-

Algorithm 4 Proxy Generation using G-MAP Framework

```
 1: Input: $\Pi, Q, B, P$, Total number of memory request J;
 2: Output: $T[][]$: Memory access sequence
 3: Determine the number of threads $K$ based on the original application.
 4: for each thread $t = 1, \ldots, K$ do
 5:     Sample $\pi_i$ from $\Pi$ with respect to $Q$.
 6:     Generate Trace $T_t$ using $\pi_i$, $B$, $P_S$ and $P_R^{(i)}$. [Algorithm 3]
 7: end for
 8: For each thread $t$ assign its corresponding warp $w$ and core $c$
 9: Perform memory coalescing for all threads in each warp.
10: Let $T_w$ denote the warp-level trace after coalescing for warp $w$.
11: For each core $c$, maintain a warp queue $WQ_c$ containing corresponding active warps.
12: while $j < J$ do
13:     for $c = 1, \ldots, MAX\_CORE$ do
14:         Choose a warp $w$ from $WQ_c$ based upon scheduling policy.
15:         $T[c][j]$ = $T_w$.get_next_access(); $j = j + 1$
16:     end for
17: end while
18: return T[][]
```

structions, G-MAP assigns memory addresses using the intra-thread stride and reuse locality information as discussed before (lines 11-18, Algorithm 3). Then, G-MAP groups individual threads into TBs and warps based on Fermi's execution model. G-MAP coalesces memory requests of threads within a warp (lines 9-10) to create coalesced warp-level traces. To model the parallel execution model of GPUs, G-MAP exploits per-core warp queues. The queue is initially filled with all active warps ordered by the warp identifier (line 11). To create a unified per-core memory access trace from the ordered per-warp traces (lines 12-17), G-MAP schedules a ready warp from the warp queue and generates an access to the memory hierarchy simulation model for the selected warp's next address (line 15). Finally, the warp queue is updated based on the warp queue maintenance policy discussed in Section

6.2.5. Miniaturization is performed by scaling down the number of proxy accesses (*J*), the intra-thread statistics and the inter-thread statistics by the desired scaling factor.

6.3 Evaluation

This section discusses the experimental setup followed by a detailed evaluation of G-MAP's performance cloning accuracy.

6.3.1 Experimental Setup

For profiling and validation, the CUDA-sim (heavily modified for profiling) and GPGPU-Sim v3.2.2 [6] simulation frameworks are used. In order to assess the efficacy of the proposed methodology across a wide variety of real-world GPU applications, 18 benchmarks, from popular GPGPU benchmark suites like Rodinia [17], NVIDIA SDK [60] and GPGPU-sim ISPASS-2009 [6], are evaluated. Each benchmark is profiled until completion or for 1 billion instructions, whichever comes first. It should be noted that profiling is a one-time cost and G-MAP receives only a statistical profile as input (independent of the execution

Table 6.2: Profiled System Configuration used for Collecting G-MAP Profiles

Component	Configuration
Core Config	15 SMs, 1400MHz, Max. 1024 Threads, 32684 Registers
L1 Cache	16KB 4-way, 128B line size, 1-cycle hit latency
L2 Cache	1MB, 8 banks, 128B line size, 8-way
Features	Memory coalescing enabled, 64 MSHRs/core, LRR sched.
DRAM	GDDR3, 8 Channels, 1 Rank/Channel, 8 Banks/Rank, 924 MHz, tRCD-tCAS-tRP-tRAS: 11-11-11-28, FR-FCFS sched. policy

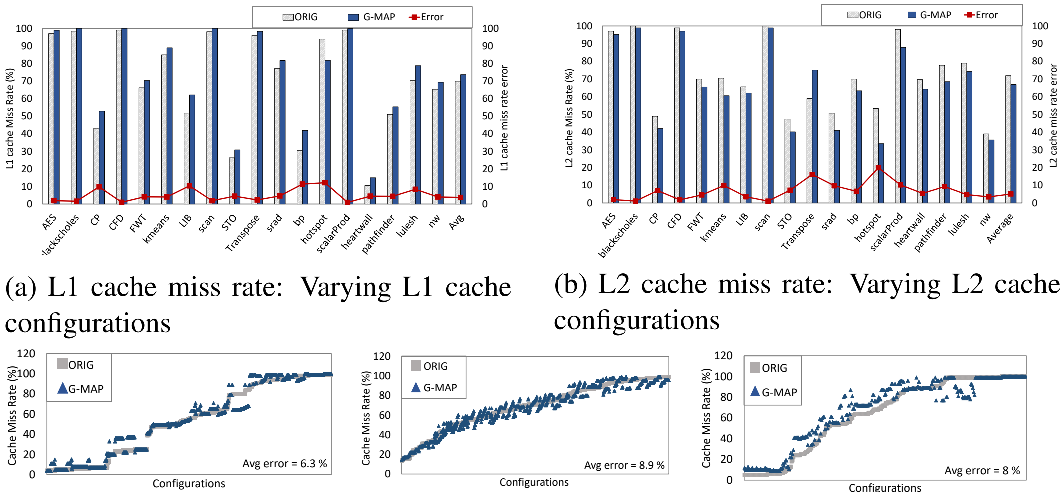
length). The 1 billion instruction interval is chosen only to keep the evaluation runs manageable. The system configuration used for collecting G-MAP profiles is shown in Table 6.2. G-MAP proxies are generated with a scaling factor of $\sim$4-5. For proxy cache and memory performance modeling, a validated SIMT-aware multi-core, multi-level cache, and memory simulator is used. The cache simulator is based on CMP$im [38]. Memory system performance is modeled using Ramulator [48], a detailed memory system simulator. G-MAP proxies are validated for modeling the performance of L1 data cache, L2 cache and the global memory system. Although G-MAP proxies are not used to evaluate the performance of shared memory or texture caches in the following experiments, G-MAP's methodology is generic enough to capture and replicate patterns in accesses to these caches as well.

6.3.2 Results and Analysis

This section evaluates G-MAP's accuracy in predicting various metrics, including the L1/L2 cache miss rates, prefetcher effectiveness and DRAM performance metrics across $\sim$290 different configurations per benchmark (over 5000 validation points in all). Specifically, two metrics are used for validation: the percentage error between original and proxy performance metrics and Pearson's correlation coefficient. Pearson's correlation coefficient indicates how well the proxies track the performance trends of the original applications (1 = perfect correlation, 0 = no correlation). For design space exploration, computer architects care about relative performance ranking; they care about comparing two configurations to see which one performs better. These two metrics together yield how closely the proxies perform with respect to the original workloads across a range of configurations.

L1 cache configurations - First, the effectiveness of G-MAP proxies is evaluated in replicating the L1 cache performance of the original applications. For this experiment, 30 different L1 configurations (varying cache size from 8 - 128KB, associativity from 1 - 16 and line-size from 32 - 128B, while keeping the L2 fixed at 1 MB, 8-way), are evaluated per benchmark, resulting in over 540 validation points across all benchmarks. The results are shown in Figure 6.6a. It can be observed that the average error between the proxy and original applications is 5.1%. Overall, G-MAP's methodology of capturing both inter- and intra-thread memory access locality leads to high accuracy across most benchmarks. For applications, such as Kmeans and heartwall, which have significant reuse locality, G-MAP's methodology of capturing and replaying reuse distance patterns leads to over 97% accuracy in mimicking L1 miss rates. Hotspot experiences the highest error because it does not have significantly dominant intra-/inter-thread stride patterns or reuse locality. Overall, the average correlation between the proxies and original applications is 0.91.

L2 cache configurations - Next, the effectiveness of G-MAP's methodology in matching the L2 cache performance of the original applications is tested (see Figure 6.6b). Here, 30 different L2 cache configurations are run per benchmark (varying the cache size from 128KB - 4MB, associativity from 1 - 16 and line-size between 64 - 128B, while keeping the L1 configuration fixed at 16KB, 4-way), resulting in over 540 validation points across all benchmarks. Overall, the average error in replicating L2 cache miss rate error is 7.1% and the average correlation is



(a) L1 cache miss rate: Varying L1 cache configurations

(b) L2 cache miss rate: Varying L2 cache configurations

(c) L1 cache miss rate: L1 prefetcher

(d) L2 cache miss rate: L2 prefetcher

(e) Cache miss rate: Diff. scheduling policies

Figure 6.6: Evaluating Cache, Prefetcher and Scheduling Policy Configurations using G-MAP Proxies: Error in Miss-rates

0.91.

L1 cache and prefetcher configurations - Regular access patterns enable prediction of future addresses, making prefetching a viable option [51, 66]. The next set of experiments test the accuracy of the memory proxies in estimating the impact of adding a state-of-the-art L1 prefetcher [51] across 72 different configurations per benchmark (varying the prefetch degree, prefetcher configurations and L1 cache configurations), resulting in over 1296 validation points. The evaluation results are shown in Figure 6.6c, sorted according to the original application cache miss rates. Overall, the average error in replicating L1 prefetcher performance is 6.3% and the average correlation is 0.9. It can be observed that the scalarProd and srad applications have regular access patterns, but they are still largely insensitive to L1 cache prefetching due to larger footprints and lower temporal locality. The hotspot application is also insensitive to prefetching because of non-dominant access patterns and low temporal locality. In contrast, the kmeans and nw applications benefit from prefetching.

L2 cache and prefetcher configurations - Next, the performance of the G-MAP proxies is compared against the original applications across different L2 prefetcher configurations. A stream prefetcher is added to the L2 cache and $\sim$96 configurations are run per benchmark (varying the stream window between 8/16/32, the prefetch degree between 1/2/4/8 and the L2 cache configurations), resulting in 1728 validation points in all. Overall, the average error in replicating L2 cache miss rate error across different cache and prefetcher configurations is 8.9% and the average correlation is 0.88 (see Figure 6.6d).

DRAM performance - Next, the effectiveness of the memory proxies is evaluated to enable design-space exploration of the memory system in lieu of the original applications. The detailed memory system, Ramulator [48], is used to evaluate 11 different GDDR5 configurations (changing the bus width, channel parallelism, DRAM addressing scheme - RoBaRaCoCh or ChRaBaRoCo) per benchmark (total 198 configurations). The memory performance of the original and proxy applications are compared over three key metrics: DRAM row buffer locality (RBL), average memory controller queue length and average read/write latency. Figure 6.7 shows the original versus clone performance values (each value is normalized with respect to the AES application's performance metrics) across the 18 benchmarks. Overall the average error in RBL, average queue length and average

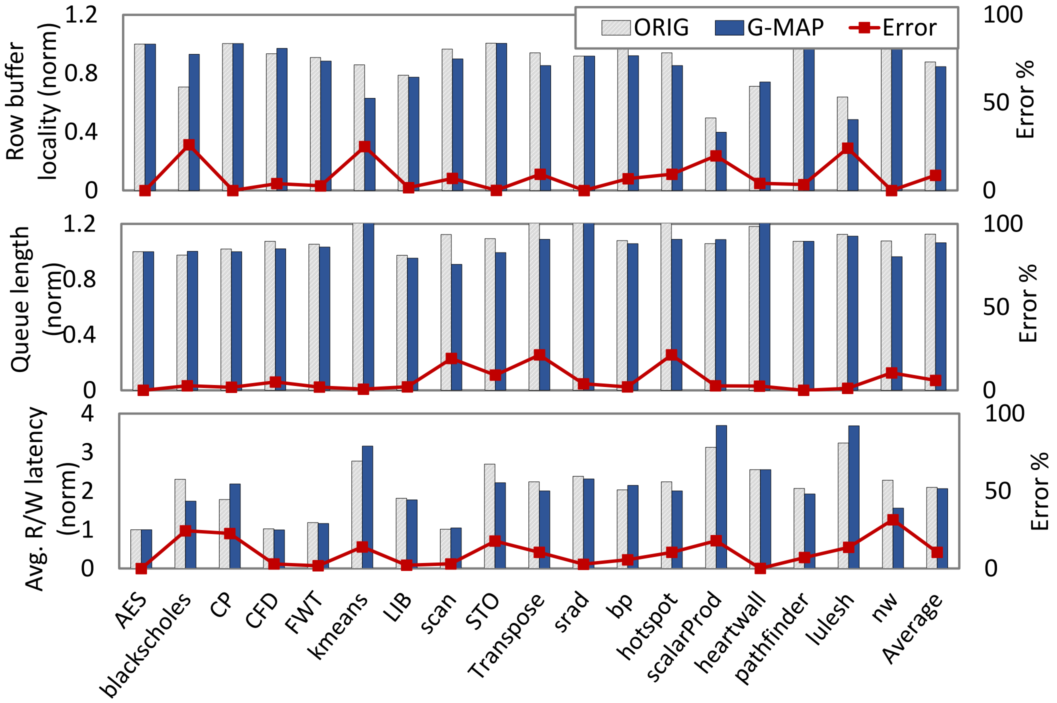


Figure 6.7: DRAM Performance Evaluation using G-MAP Proxies

read-write latency are 9.95%, 8.64% and 12.6%, respectively (average correlation = 0.85).

Scheduling policy impact - The next set of experiments test the effectiveness of G-MAP's methodology in replicating cache and memory performance across two scheduling policies, Greedy-then-oldest (GTO) and LRR (see Figure 6.6e). As discussed before, G-MAP does not model the GPU cores and it adopts an approximate policy to schedule threads. Nevertheless, the average error in replicating L1 cache miss rate is 8% (5.1% for LRR and 10.9% for GTO policy).

Impact of trace miniaturization - Since G-MAP relies on statistical convergence to replicate memory access patterns, it is important to have sufficient num-

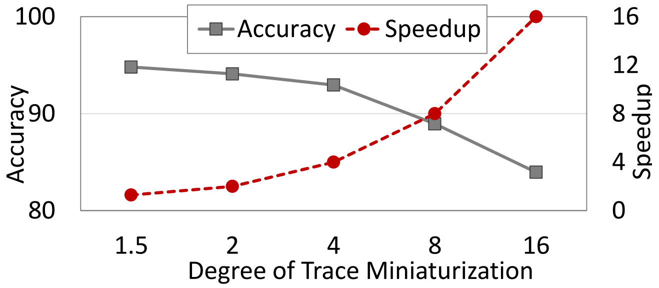


Figure 6.8: Impact of Trace Miniaturization

ber of samples in the original application to replicate the different probability values due to the law of large numbers. Figure 6.8 shows the impact of higher degree of trace miniaturization on the performance cloning accuracy (left axis) and speedup of memory simulation using the reduced clone over the full trace (right-axis). It can be observed that, as the trace size is reduced, the simulation speed increases almost linearly, while the performance cloning accuracy starts dropping after a certain point. At 8x trace size reduction, the accuracy drops to $\sim$90%, while the simulation speed improves by $\sim$8x. The degree of miniaturization on real-world applications can be higher since the number of samples in the real-world application memory traces is often very large.

6.4 Summary

This chapter presented G-MAP, a novel methodology to statistically model the memory access behavior of GPU applications by leveraging the synergy in code-localized access patterns within and across threads. G-MAP also accounts for GPU's parallel execution model by adopting a fine-grained, coordinated scheduling policy to ensure appropriate degree of parallelism at the thread-level and the cache/memory-level. Extensive evaluations using over 5000 different cache, memory and prefetcher configurations and 18 different GPGPU benchmarks are performed to show that G-MAP proxies can replicate the cache/memory performance of the original GPU applications with over 90% accuracy, while significantly reducing the simulation time and storage requirements.

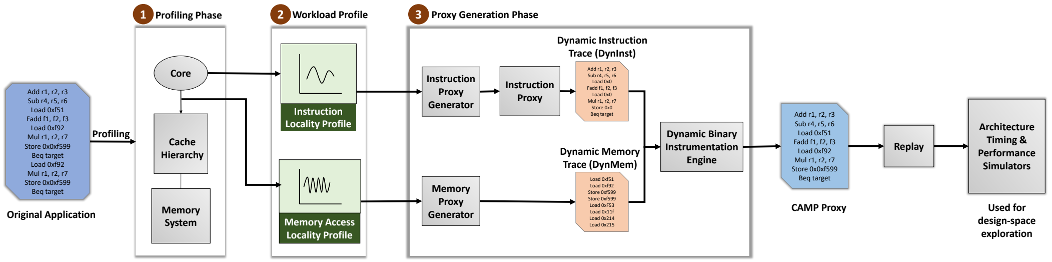
Chapter 7

CAMP: Accurate Modeling of Core and Memory Locality for Proxy Generation of Big-data Applications

Prior system-level proxy generation proposals [42, 29, 69] model core-level locality metrics in detail, but abstract out memory locality modeling using very simple dominant stride-based models. This results in poor cloning accuracy of the proxy benchmarks, especially in applications with complex memory access patterns [7, 5]. Most big-data applications are highly data-intensive and their overall system-level performance is significantly impacted by the performance of the cache and memory hierarchy [26, 67, 104]. As a result, prior system-level performance cloning techniques are not very effective or accurate at studying the overall performance of big-data applications. A few detailed cache and memory cloning techniques [7, 5] have also been proposed in literature. For example, STM [5] tracks long history-based stride transitions in the global memory reference sequence of applications to generate miniature memory clones. Such techniques generate only a memory access trace, which can be used for cache/memory hierarchy design space exploration, but do not model any core or instruction locality behavior.

In reality, the processor core configuration and the application together determine processor performance, which affect the timing of requests received in the memory system. At the same time, memory performance has a feedback loop with processor performance, which in turn affects timing of other memory requests and overall behavior of the application. None of the prior cloning studies accurately model the joint performance of both core and memory subsystems and their complex interactions. As such, there is need for system-level proxy benchmarking techniques that can model both core and memory performance accurately.

This chapter focuses on synthesizing accurate and representative proxy benchmarks to study system-level (core and memory) performance of emerging applications. This chapter introduces **CAMP**, a novel proxy generation and modeling methodology that accurately models both **C**ore performance **A**nd **M**emory locality to create miniature **P**roxy benchmarks. CAMP proxies do not need any complex software-stack support and have shorter execution times [73]. To model the core performance, CAMP adopts existing methods for generating proxy instruction streams by capturing and modeling the dependencies between instructions (instruction-level parallelism), instruction types, control-flow behavior, etc. CAMP also adds an improved memory locality profiling approach that captures both the spatial and temporal locality of applications. However, most big-data applications typically do not have a single dominant stride/offset based access pattern [67, 70, 104]; thus, it is quite challenging to control the different dynamic execution instances of the low-level, static load/store instructions in the proxy benchmark to reproduce the complex memory access patterns of the original applications using synthetic data-structure accesses in the proxy code. To address this challenge, CAMP introduces a novel proxy generation and modeling/replay methodology that


1 Profiling Phase
2 Workload Profile
3 Proxy Generation Phase
Original Application
Profiling
Core
Cache Hierarchy
Memory System
Instruction Locality Profile
Memory Access Locality Profile
Instruction Proxy Generator
Instruction Proxy
Dynamic Instruction Trace (DynInst)
Memory Proxy Generator
Dynamic Memory Trace (DynMem)
Dynamic Binary Instrumentation Engine
CAMP Proxy
Replay
Architecture Timing & Performance Simulators
Used for design-space exploration

```
Add r1, r2, r3
Sub r4, r5, r6
Load 0xf51
Fadd f1, f2, f3
Load 0xf92
Mul r1, r2, r7
Store 0x0xf599
Beq target
Load 0xf92
Mul r1, r2, r7
Store 0x0xf599
Beq target
```

```
Add r1, r2, r3
Sub r4, r5, r6
Load 0x0
Fadd f1, f2, f3
Load 0x0
Mul r1, r2, r7
Store 0x0
Beq target
```

```
Load 0xf51
Load 0xf92
Store 0xf599
Store 0xf599
Load 0xF53
Load 0x11f
Load 0x214
Load 0x215
```

```
Add r1, r2, r3
Sub r4, r5, r6
Load 0xf51
Fadd f1, f2, f3
Load 0xf92
Mul r1, r2, r7
Store 0x0xf599
Beq target
```

Figure 7.1: CAMP's Profiling and Proxy Generation Framework

integrates the core and memory locality models to create accurate system-level proxy benchmarks. This approach enables CAMP proxies to mimic the original application's core as well as memory performance behavior and capture the performance feedback loop between core and memory subsystem well. For a variety of real-world database applications, CAMP achieves an average cloning accuracy of over 89%. This novel proxy benchmarking capability can facilitate evaluation of overall system (core, cache and memory subsystem) design-space exploration.

7.1 CAMP's Methodology

Figure 7.1 shows an overview of CAMP's core and memory locality modeling framework. During the profiling phase ①, CAMP characterizes the inherent instruction (e.g., instruction-level parallelism, instruction mix) and memory access locality patterns (e.g., spatial & temporal locality of memory accesses) of big-data applications to create a statistical workload-specific profile ②. During the proxy generation and modeling phase ③, CAMP adopts a systematic methodology to create a miniaturized clone of the big-data application based on the workload-specific profile, which can be used to drive CPU core, cache & memory performance exploration. Next, CAMP's workload characterization methodology is discussed, followed by the proxy generation and modeling algorithm.

7.1.1 Workload Profiling

CAMP's profiling infrastructure (see ① in Figure 7.1) is implemented over the micro-architectural processor simulator, MacSim [47]. The profiler modules are developed separately from the simulator's original code as stand-alone modules. To characterize the original big-data application and extract the workload-specific statistics (details in Table 7.1), profiling probes are inserted into the simulation infrastructure at two points - one before the decode stage of the processor pipeline to collect the "**instruction locality profile**" and another before the data cache access ports to collect the "**memory access locality profile**". Next, the different characteristics collected corresponding to the instruction and memory locality profiles will be discussed.

7.1.1.1 Instruction Locality Parameters

Following are the different instruction locality features captured by CAMP (see Table 7.1 for a profile summary).

a. Basic-block features and instruction footprint - CAMP identifies the number of dominant static basic blocks in the original big-data application, which constitute a fixed threshold (empirically, set to 90% in our case) of the big-data application's total dynamic basic-block count. The number of basic blocks instantiated in the proxy benchmark is set to the number of dominant basic blocks

Table 7.1: CAMP's Profiled Statistics

Statistic	Description
f_{imix}	Instruction mix distribution (e.g., #loads, #branches)
$P_{\delta_1,\delta_2,...,\delta_{128}}$	Dependency distance distribution $(1,2,\ldots,\leq 128)$
P_{BrTr}	Branch transition frequency distribution (0-100%)
f_{sys}	Fraction of system activity
B	Number of basic blocks in the proxy
B_{size}	Average basic block size
SSD_{ij}	Stack distance probability at the i^{th} set and j^{th} stack position
$SHT_{\{s_1,s_2,...,s_i\}->nstrs}$	Stride pattern table keeping stride transition counts from past i strides to next strides (nstrs)
S_i	Fraction of accesses to the i^{th} set
W_c	Probability of write to clean block
W_d	Probability of write to dirty block
ρ_w	Fraction of write accesses

identified in the original application. A lower threshold value can lead to a higher degree of miniaturization, but comes at the expense of a loss in cloning accuracy. Next, CAMP tracks the average basic block size of the dominant basic blocks and transition probabilities between pairs of basic blocks. Average basic block size is an important metric because it determines the average number of instructions that can be executed in the program sequence without executing any control instructions. This can affect performance significantly depending on the branch predictor performance.

b. Instruction mix - The next set of metrics captured by CAMP is the instruction mix of the original application. The instruction mix (IMIX) of a program measures the relative frequency of various operations performed by the program and is an important determinant of an application's performance. For example, an integer division operation often takes more cycles to execute than simpler arithmetic instructions. The fraction of floating-point and integer instructions influences a program's execution time. CAMP measures the IMIX of the big-data applications, specifically in terms of the fraction of integer arithmetic, integer multiplication, integer division, floating-point operations, SIMD operations, loads, stores and control instructions in the dynamic instruction stream of the program. The captured IMIX statistics are used to populate corresponding instructions into the static basic blocks in the proxy benchmark.

c. Control-flow behavior - Another important metric that affects big-data application performance is its control flow behavior. Difficult-to-predict control instructions and irregular branch behavior lead to poor branch predictor performance, which causes higher number of wrong-path executions, pipeline flushes and degrades overall performance. Prior research work [43, 11, 29] has shown that an application's branch misprediction rate is highly correlated with the transition frequency of the branch instructions [34]. Branch transition rate measures how often a branch transitions between its taken and not-taken paths and is an indicator of the overall predictability the branch instructions. CAMP captures the transition rate of the branch instructions in the big-data applications and bins them into eight buckets, where each bucket represents the fraction of control instructions having a transition rate ranging from 0-100%. To model a certain branch transition rate in the proxy benchmark, each branch instruction is assigned a transition frequency to satisfy the overall target branch transition rate.

d. Instruction-level parallelism - Next, CAMP captures the instruction-level parallelism (ILP) of the original big-data applications. Tight producer-consumer based dependency chains can significantly affect application performance due to serialization effects. CAMP models an application's ILP based on its inter-instruction dependency distance, which is defined as the number of dynamic instructions between the production (write) and consumption (read) of a register/memory operand. Figure 7.2 shows a simple dependency distance computation for an example program fragment, with a true read-after-write dependency between the first and fourth instructions. CAMP classifies the instruction dependency distance into eight bins $(1,2,\leq 4,\leq 8,\ldots,\leq 128)$, where each bin represents the percentage of instructions having that particular dependency relation. During proxy benchmark generation, an instruction's register or memory operands are assigned a dependency distance to satisfy the dependency metrics collected from the original application.

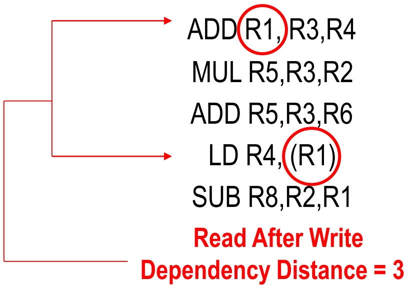


Figure 7.2: Dependency Distance Computation Example

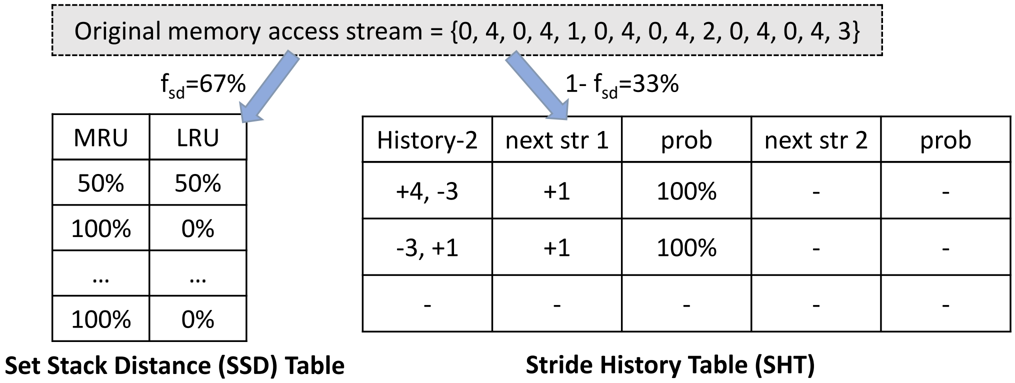
e. System activity - Many emerging, big-data applications spend a significant fraction of their execution time executing operating system code [26, 67]. To model the impact of high system activity, CAMP tracks the fraction of executed user-mode and kernel instructions in the big-data applications during profiling. Next, CAMP adds the target fraction of system calls into the proxy benchmark during proxy generation.

7.1.1.2 Memory Locality Parameters

As discussed, prior system-level proxy benchmarking techniques use a very simple model to capture memory access locality. They model locality of individual load/store instructions in the original application based on a single dominant stride value. Although such an approach can work for small loop-based programs (e.g., array-based accesses), the memory instructions in most big-data applications have quite random, complex access patterns which cannot be captured by a single stride alone [26]. For example, join operations using hash tables, key-value stores and complex structures such b-trees do not lend themselves well to dominant strides as a representative model [67]. This section will discuss how CAMP addresses the need for a more representative memory model.

Different requests in the cache and memory subsystem are generated for the following reasons: (a) memory read-write requests are caused by actual load/store instructions in an application, (b) speculative prefetch requests are typically generated by a hardware prefetching engine and (c) write-back requests are generated by the upper level caches (e.g., write-back caches) and sent to the lower levels of the cache/memory hierarchy. While the first type of requests are generated by execution of instructions on the processor, the other two depend on the architecture (e.g., cache write-policy, prefetcher configuration).

To accurately model the read requests generated by memory instructions run by the core (type (a)), CAMP captures the per cache-set stack distance distribution (SSD) profile [55] for a baseline L1 cache configuration (16KB, 2-way). The SSD profile captures the fraction of memory references to the different LRU stack positions (stack position 0 represents the most recently used block, stack position 1 represents the second most recently used block, etc.) within every cache set of the baseline L1 cache. For example, SSD_{ij} represents the probability for an access to fall in the i^{th} set at the j^{th} LRU stack position. Using SSDs helps to capture the temporal locality of memory access streams. For the accesses that miss in the L1 cache, their spatial locality patterns are captured by learning global stride transitions in a stride history table (SHT). A stride is defined as the difference between addresses of two consecutive memory accesses, which miss in the profiled L1 cache. Each SHT entry records a history of past consecutive stride values (length is based the history depth), and the next strides that followed the stride history in the past, and each next stride's frequency of occurrence. CAMP also collects another statistic, f_{SD}, which records the fraction of memory accesses that hit in the SSD table. Figure 7.3 illustrates a simplified view of these profiling structures. Note that these statistics are the only ones that are similar to the statistics used in STM [5] and all the remaining statistics are unique to CAMP. Additionally, it should be noted that although CAMP leverages STM's cloning technique for memory locality modeling, any other memory locality modeling technique (such as HALO or WEST) can be used instead of STM to generate the proxy memory address traces. CAMP's novelty lies in how to integrate a core and memory locality model together to create


Original memory access stream = {0, 4, 0, 4, 1, 0, 4, 0, 4, 2, 0, 4, 0, 4, 3}
f_{sd}=67%
1- f_{sd}=33%

MRU	LRU
50%	50%
100%	0%
…	…
100%	0%

Set Stack Distance (SSD) Table

History-2	next str 1	prob	next str 2	prob
+4, -3	+1	100%	-	-
-3, +1	+1	100%	-	-
-	-	-	-	-

Stride History Table (SHT)

Figure 7.3: STM-based Memory Locality Profiling

more accurate system-level proxy benchmarks.

Apart from tracking the per-set SSD profiles (like STM), it is equally important to capture the distribution of memory accesses across sets. Not capturing access distribution across sets leads to different conflict behavior between cache sets when L1 test configurations differ from the baseline, resulting in cloning errors. So, in addition to tracking per-set SSD profiles, CAMP also captures the fraction of accesses (S_i) that are generated to every set of the baseline L1 cache. Together, the above profiles provide sufficient temporal and spatial locality information to replicate the behavior of processor memory requests and prefetch requests.

However, STM's statistics are not sufficient to deal with write-back request traffic in the memory system. STM collects a statistic, ρ_w, which records the fraction of accesses in the original application that are writes. However, the number of write-backs is not determined by the number/fraction of write accesses. Rather, it is dependent on the number of dirty cachelines in the cache hierarchy and this is not captured by STM's write statistics. Figure 7.4 shows an example scenario, which leads to different number of write-back requests in STM versus the original application. In the original program, 50% of the accesses are stores, but the stores happen to the same cache block, resulting in one dirty block in the cache. As STM does not know about the distribution of stores to clean or dirty cachelines, it can generate two writes to different blocks, resulting in two dirty blocks and two future write-back requests. In order to capture this effect, CAMP records the number of writes to clean and dirty blocks. When a clean block receives a write request, it becomes dirty and subsequent read/write operations on the same block do not impact its dirty status. Based on the counts aggregated during the profiling phase, two probabilities are computed, W_c and W_d, which represent the probability of writing to a clean or dirty block, respectively. During proxy synthesis, the request type (load or store) is selected based on the clean or dirty state of the generated address and the conditional probabilities (W_c and W_d).

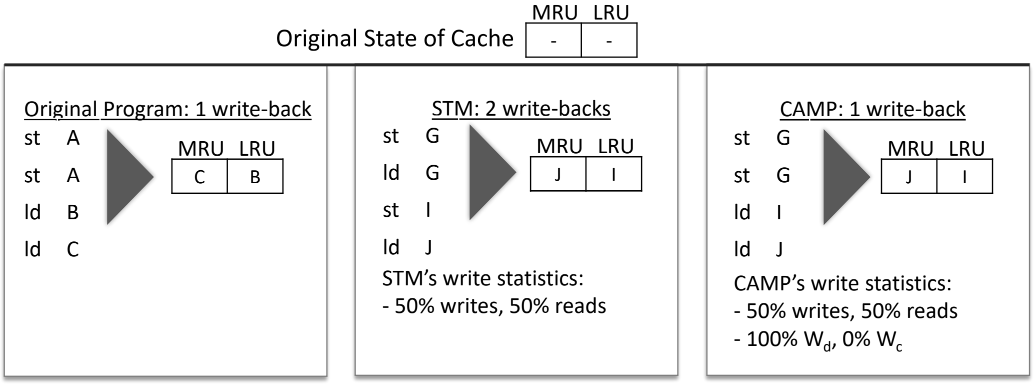


Figure 7.4: Profiling for Write-back Requests

7.1.2 Proxy Generation and Modeling

In this section, a detailed description of CAMP's proxy generation process (See ③ in Figure 7.1) will be provided. Table 7.1 summarizes the captured locality metrics.

Algorithm 5 shows how the "memory proxy generator" leverages the memory locality profiles to create a dynamic memory access trace (*DynMem*). First, the number of memory addresses accessed in the proxy (N) is estimated after applying the desired degree of miniaturization. For each memory access, CAMP chooses whether it will generate a proxy address using the SSD or the SHT profiles depend-

Algorithm 5 CAMP's Dynamic Memory Proxy Generation Algorithm

```
**Output:** $DynMem[] = \{(ADDR_1, RW_1), \ldots, (ADDR_N, RW_N)\}$;
**for** $n = 1, \ldots, N$ **do**
    Sample $f_n \in \{0, \ldots, 100\}$;
    **if** $f_n$ ¡ $f_{SD}$ **then**
        Use $S_i$ to choose set and $SSD_{ij}$ to choose stack distance position;
        Choose $ADDR_n$ based on chosen set and stack position;
    **else**
        Sample stride $S_n$ from $SHT$ based on $LAST\_STR$;
        $ADDR_n = LAST\_ADDR + S_n$;
        $LAST\_STR$.push_back($S_n$); $LAST\_ADDR = ADDR_n$;
    **end if**
    **if** $ADDR_n \in DirtyBlocks$ **then**
        Sample $W_d$ and $\rho_w$ to assign $RW_n$;
    **else**
        Sample $W_c$ and (1-$\rho_w$) to assign $RW_n$;
    **end if**
    $DirtyBlocks.add(ADDR_n)$ if $RW_n = Write$;
**end for**
**return** DynMem[]
```

ing upon the f_{SD} probability. If the SSD profile is used, CAMP picks the address located at a set and way chosen using the SSD_{ij} and S_i profiles (line 6). If the SHT profile is used, then the next address is chosen based on the stride transitions saved for the current stride history ($LAST\_STR$, lines 8-9). To make a load/store assignment, CAMP checks if the chosen address block is dirty or clean. Accordingly, it uses W_d or W_c (and ρ_w) to determine if the instruction should be a load or store (lines 12-17). This process is repeated till the target number of memory accesses are generated. The resultant trace forms the *DynMem* proxy trace.

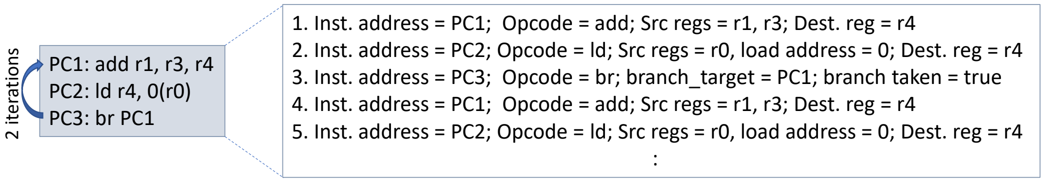
Next, the "instruction proxy generator" leverages the instruction locality metrics to create an instruction proxy (see Algorithm 6). First, CAMP populates each basic block in the proxy with an appropriate number (satisfying the mean and standard deviation of the target B_{size}) and type (satisfying f_{imix}) of instructions.

Algorithm 6 CAMP's Instruction Proxy Synthesis Algorithm

```
 1: Output: Instruction proxy, B[]
 2: while $b < B$ do
 3:     Sample a random basic block based on its access frequency.
 4:     Find basic-block size $I$ to satisfy mean & std. dev of target $B_{size}$;
 5:     for $i < I$ do
 6:         Assign instruction type based on target $f_{imix}$;
 7:         Assign dependency relation based on target $P_\delta$ distribution;
 8:         For memory ins., assign a 0 stride to base array;
 9:         Inject system-calls based on target $f_{sys}$;
10:         Insert x86 $test$ operation with chosen modulo operand.
11:         Assign last instruction to be conditional branch instruction.
12:     end for
13: end while
14: Assign architectural register operands to satisfy dependency relations of step 7.
15: return B[]
```

The last instruction of every basic block is instantiated as a conditional branch instruction. Next, each instruction is assigned a dependency distance (i.e., a prior instruction that generates its operands) to satisfy the dependency distance criterion (line 8). As the memory instructions in most big-data applications typically do not have a fixed stride/offset, it is very challenging to control the different dynamic execution instances of the low-level, static load/store instructions in the instruction proxy in order to dynamically produce the same dynamic memory access sequence produced by Algorithm 5 (*DynMem*). To achieve this, all the memory instructions in the instruction proxy are temporarily assigned to have a zero stride with respect to a baseline array. After instruction proxy generation completes, a binary instrumentation engine is used to integrate the *DynMem* trace into the instruction proxy (details will be discussed in the next paragraph). Next, system calls are injected (or not) into the basic block based on the target system-call frequency (line 10). An x86 test operation is inserted before every branch to set the condition codes that control the branch's outcome. The test instruction's operand is chosen to control the transition frequency of the branch instruction (line 11) to achieve the target transition rate of every basic block. The above steps are repeated till the target number of basic blocks (*B*) are generated. Finally, architectural registers are assigned to each instruction to satisfy the identified dependencies. The instruction proxy generator generates C-language based proxies with embedded x86-based assembly instructions using the *asm* construct. The proxy instructions are nested under a two-level loop where the loop iterations control the number of dynamic instructions.

After the static instruction proxy program is created, a binary instrumen-



```
PC1: add r1, r3, r4
PC2: ld r4, 0(r0)
PC3: br PC1
```

```
1. Inst. address = PC1;  Opcode = add; Src regs = r1, r3; Dest. reg = r4
2. Inst. address = PC2; Opcode = ld; Src regs = r0, load address = 0; Dest. reg = r4
3. Inst. address = PC3;  Opcode = br; branch_target = PC1; branch taken = true
4. Inst. address = PC1;  Opcode = add; Src regs = r1, r3; Dest. reg = r4
5. Inst. address = PC2; Opcode = ld; Src regs = r0, load address = 0; Dest. reg = r4
                                        :
```

(a) Instruction proxy

(b) Dynamic instruction proxy stream (DynInst)

Figure 7.5: DynInst Format

tation tool (e.g., PIN) is used to generate a dynamic instruction stream (*DynInst*) of the same. An example format of *DynInst* is shown in Figure 7.5. Next, a dynamic binary instrumentation engine is implemented for integrating the *DynMem* and *DynInst* profiles to create the unified CAMP proxies, capturing both instruction and memory access behavior of the original big-data applications. For every dynamic execution instance of load/store instruction in the *DynInst* sequence, the instrumentation engine overrides the temporary address assigned to the instruction during the instruction proxy generation time with the next address from the *DynMem* sequence in a serialized fashion. The instrumented instruction and memory stream forms the CAMP proxy. To evaluate the CAMP proxies on simulators, a replay engine is implemented that interfaces with the trace reading logic of the architectural performance or power simulator and feeds the simulator with the unified CAMP dynamic instruction and memory sequences. Most architectural simulators (e.g., SniperSim [15], MacSim [47], Ramulator [48]) support well-defined dynamic trace driven execution modes and CAMP proxies could be easily integrated into such frameworks by modifying the replay engine.

7.2 Evaluation

This section discusses the experimental setup followed by a detailed evaluation of CAMP's performance cloning accuracy.

7.2.1 Experimental Setup

For profiling and validation, a detailed architecture simulator, MacSim [47] is used. CAMP is evaluated using a set of big-data data-serving (Yahoo! Cloud Serving Benchmark (YCSB)[18]) and data-analytics applications (TPC-H benchmarks [100]). The standard benchmarks provided with YCSB framework are run. These benchmarks cover the most important operations (read, write, insert and scan) performed against a typical data-serving database. TPC-H models a decision-support system for order-processing engines, with queries performing different business-like analyses. CAMP is evaluated using 5 TPC-H benchmark queries. Both the TPC-H and YCSB benchmarks interact with a back-end MySQL database. The test databases are chosen to have a total size of $\sim$10-12GB so that the data fits into memory of the server nodes, which is the recommended operational setup for scale-out applications for better performance [57]. Each benchmark is executed by

Table 7.2: Profiled System Configuration used for Collecting CAMP Profiles

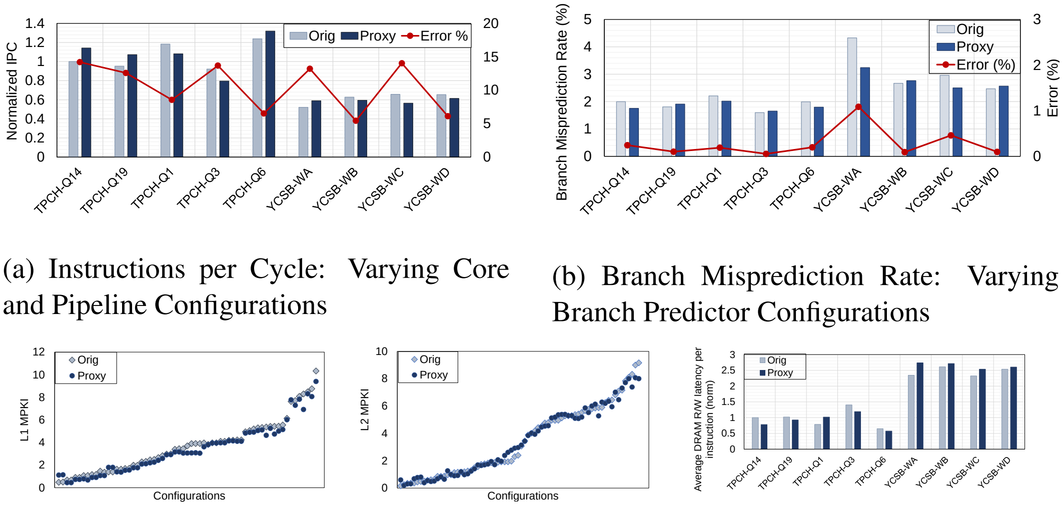
Component	Configuration
CPU	x86_64 processor, atomic mode, 4 GHz
L1 Cache	32KB, 2 way Icache; 64KB, 2 way Dcache; 64B block size, LRU
L2 Cache	256KB, 4-way, LRU
DRAM	16GB DDR3, 12.8 GB/sec

first fast-forwarding to skip the initialization stage and then, cloning one particular phase of the application consisting of 1 billion instructions (to capture other phases, other 1 billion instruction windows can be chosen). It should be noted that profiling is a one-time cost and CAMP receives only a statistical profile as input (independent of the execution length). The system configuration used for collecting CAMP profiles is shown in Table 7.2.

7.2.2 Results and Analysis

This section evaluates CAMP's accuracy in predicting various performance metrics across different core, pipeline, branch predictor, cache and memory configurations.

Core configurations - First, the effectiveness of CAMP proxies in replicating overall performance of the database applications is evaluated. For this experiment, 8 different core configurations are run per benchmark by changing the pipeline width between 2-8, re-order buffer size between 128-512 and issue rate between 2-4. Figure 7.6a shows the results. It can be observed that the average error between the proxy and original applications is $\sim 11\%$. The highest error is experienced by the TPCH-Q14 benchmark as it suffers from aliasing effects in the stride history table due to complex join-based access patterns, leading to higher L1/L2 cache and memory performance cloning errors. Increasing the stride history length can improve memory performance cloning accuracy, but comes at the expense of higher metadata overhead. Overall, CAMP's methodology of capturing different instruction and memory access locality metrics leads to small error rates



(a) Instructions per Cycle: Varying Core and Pipeline Configurations

(b) Branch Misprediction Rate: Varying Branch Predictor Configurations

(c) L1 MPKI: L1 Cache and Prefetcher

(d) L2 MPKI: L2 Cache and Prefetcher

(e) DRAM R/W latency: Diff. DRAM Configs

Figure 7.6: Evaluating Core, Branch Predictor, Cache, Prefetcher and DRAM Configurations using CAMP Proxies

across most benchmarks (including complex queries in TPC-H and YCSB benchmarks). Overall, the proxies replicate the overall performance behavior with 0.94 correlation.

Branch predictor configurations - The next set of experiments evaluate the effectiveness of CAMP proxies in replicating behavior of original applications across different branch predictor configurations. In particular, two different branch predictors (gshare and tournament) are tested and the predictor's branch history depth is also varied between 8 - 18. Figure 7.6b shows the average error in branch misprediction rate between the original and proxy applications. It can be observed that the average error is less than 1% (correlation = 0.93). This shows that CAMP's methodology of using branch transition rates to track predictability of control instructions is fairly accurate at modeling application behavior across different branch predictor configurations.

L1 cache and prefetcher configurations - Next, CAMP's cloning effectiveness is validated across different L1 cache and prefetcher configurations. In particular, 6 different L1 cache configurations are evaluated per benchmark (varying the cache size from 16KB-64KB and the associativity from 2-8). For each cache configuration, the L1 stream prefetcher configuration is also varied by changing the stream detection window between 8-16 and prefetch degree between 0-4. Results showing the L1 cache miss rate errors are shown in Figure 7.6c. Capturing both temporal and spatial locality patterns using long history-based stride transitions in the memory access streams of the complex, big-data applications leads to highly accurate replication of cache performance. The proxies experience about an error of up to 2 MPKI in some configurations, especially when the cache line size of the L1 caches changes significantly because the collected memory profiles do not capture locality within cache-blocks. Nonetheless, it can be observed that the overall correlation between the proxy and the original applications is high (0.98).

L2 cache and prefetcher configurations - The next experiment evaluates CAMP's effectiveness across different L2 cache and prefetcher configurations. Eight different L2 cache configurations are evaluated per benchmark (varying the cache size from 128KB-1MB and the associativity from 2-16). For each cache configuration, the L2 stream prefetcher configuration is also varied by changing the stream detection window between 8-16 and the prefetch degree between 0-2.

Results showing the L2 cache miss rate errors are shown in Figure 7.6d. It can be observed that the overall correlation is high (0.97) due to accurate modeling of load-store patterns and write-back cache traffic.

DRAM performance - Next, the effectiveness of CAMP proxies at enabling design-space exploration of the memory system in lieu of the original applications is verified. In particular, 6 different DRAM configurations are evaluated (changing the memory controller scheduling policy between FR-FCFS and FCFS, channel parallelism between 4-8 and the row buffer size between 2-4KB) per benchmark (resulting in 54 total data-points). The original and proxy benchmarks are compared in terms of average read/write latency per instruction (see Figure 7.6e). Each value is normalized with the original TPC-H Q14 benchmark's performance metrics. Overall the average error in average read-write latency per instruction is 14.5% (average correlation = 0.89).

Comparison with prior techniques - Figure 7.7 contrasts the result of clones generated using the single dominant stride (SDS proxy) profile, the most commonly used statistic in literature for modeling memory locality patterns in system-level proxy benchmarks, against CAMP proxies. For this, the L1 cache size is varied from 16-64KB and associativity from 2-8 ways. It can be observed that the SDS clones show significant errors in miss ratio (measured as the absolute difference in miss ratios) at many data points, reaching as high as 33%. CAMP proxies, on the other hand, show near-zero errors across almost all cases, and only shows a few data points with relatively elevated error ($\leq$ 4%). This result demonstrates that the SDS approach is not suitable for modeling the complex memory

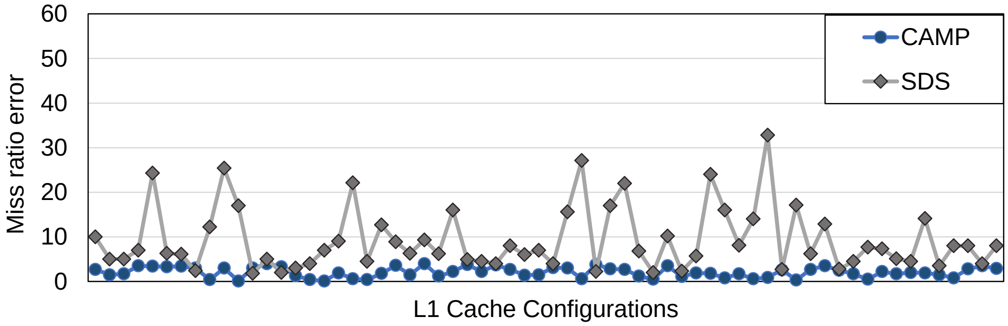


Figure 7.7: Comparing L1 Miss-ratio Cloning Error Between CAMP and SDS Proxies for Various L1 Cache Configurations.

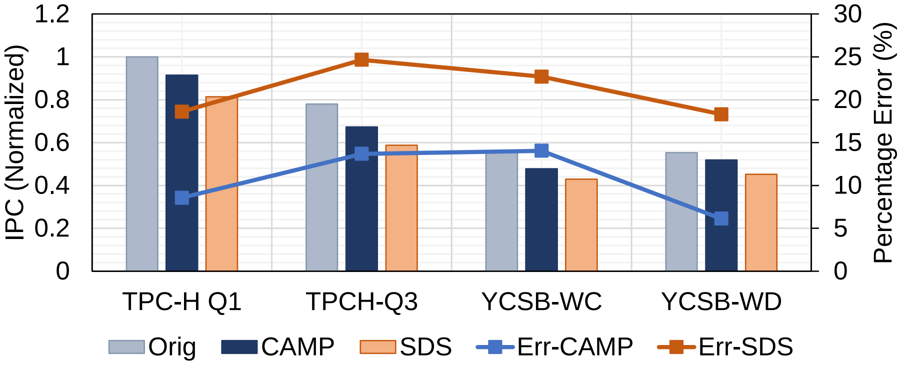


Figure 7.8: IPC Cloning Accuracy of CAMP versus SDS proxies.

access patterns of big-data applications. Figure 7.8 compares the IPC cloning accuracy of the SDS and CAMP proxies for three big-data benchmarks across different core pipeline and cache configurations. Overall, it can be observed that accounting for accurate memory locality models together with replicating the program ILP, instruction types, basic blocks, etc. helps CAMP proxies to achieve much lower cloning error ($\sim 11\%$) compared to SDS proxies ($\sim 21\%$).

Degree of miniaturization - Since CAMP relies on statistical convergence to replicate instruction and memory locality, it is important to have sufficient number of samples in the original application to replicate the different probability values due to the law of large numbers. The proxies contain roughly 90-100 million dynamic instructions, yielding a clone that is roughly 10-12x smaller than the original application and as a result, achieving a $\sim$10x reduction in simulation time. The degree of miniaturization on real-world applications can be higher since the number of samples in the real-world application traces is often very large. Further miniaturization can be achieved by reducing the number of instructions, but at the expense of a drop in the cloning accuracy.

HALO memory model - CAMP is a very flexible system-level proxy benchmark generation technique that allows combining a instruction-pipeline locality modeling technique with an accurate memory locality modeling technique to create miniature system-level proxy benchmarks. This section replaces the STM-based memory model, which was used in previous experiments, with a HALO-based memory model. This experiment requires collecting HALO-specific memory locality profiles (as discussed in Chapter 5). The other components of CAMP (as shown in Figure 7.1) remain unchanged. Figure 7.9 compares the IPC of the original and proxy benchmarks across different core configurations (pipeline width is varied between 2-8, re-order buffer size is varied between 128-512 and issue rate is varied between 2-4). It can be observed that the average error between the proxy and original applications is $\sim$10%. As discussed earlier, HALO out-performs STM in modeling cache and memory reference locality. As a result, using HALO based memory model leads to slightly more accurate proxy benchmarks. This experiment

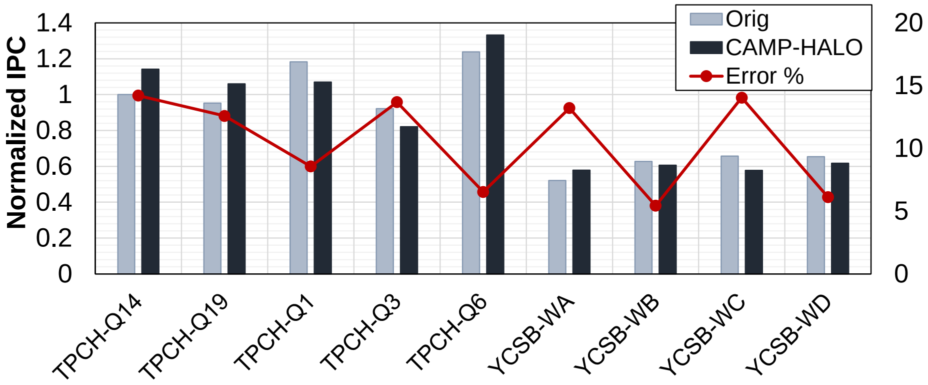


Figure 7.9: IPC Cloning Accuracy of CAMP-HALO proxies versus Original Workloads.

however intends to demonstrate that CAMP's proxy generation, modeling and replay methodology is very generic.

7.3 Summary

This chapter presented CAMP, a system-level proxy benchmarking technique to solve the confidentiality and representativeness problems of workload performance cloning for big-data applications. CAMP models both core-performance and memory locality accurately along with modeling the feedback loop between the core and memory performance. To model the core performance, CAMP adopts existing methods for generating proxy instruction streams by capturing and modeling the dependencies between instructions (instruction-level parallelism), instruction types, control-flow behavior, etc. An improved memory locality profiling approach is added that captures both the spatial and temporal locality of applications. Finally, a novel proxy generation and replay technique is used to integrate the core and memory locality models together to create accurate system-level proxy benchmarks. It was demonstrated that CAMP clones mimic the original application's performance behavior and that they capture the performance feedback loop well. For a variety of real-world database applications, CAMP proxies achieved an average cloning error of $\sim$11%. This system-level proxy benchmarking technique is expected to be a new capability that can enable accurate overall system (core and memory subsystem) design-space exploration.

Chapter 8

Synthetic Workload Generation using Proxy Generator Framework to Densely Cover Performance Spectrum

During a typical computer system design cycle (often spanning over several years), applications can change quite significantly; for example, new applications are introduced, existing applications are expanded to work on bigger problems and data footprints, and even the same applications may produce different workloads due to better compiler optimization. However, it is impractical to create a new standard benchmark whenever a new application domain or software paradigm emerges. For example, the latest SPEC CPU benchmark suite, SPEC CPU2017 is just released after a gap of 11 years [72]. As a result, computer designers traditionally rely on benchmarks of yesterday or today to build machines for tomorrow. Another issue is that the set of programs included in a standard benchmark suite is fixed and the benchmarks often fill only certain data points in the workload design-space. Much of the workload design-space map is not represented by any workloads. It will be useful to have broader and denser coverage of the workload map. This chapter introduces Genesys [75], a methodology to create hypothetical benchmarks, by tweaking program characteristics in a systematic way to produce new workloads with performance behavior that does not currently exist for filling up the workload performance spectrum more densely.

8.1 Genesys's Methodology

Genesys is a synthetic workload generation framework (shown in Figure 8.1) that enables systematic generation of synthetic applications covering a broad range of the application state-space. It builds upon a set of key workload-specific metrics that can be controlled systematically to generate workloads with desired properties. Each workload-specific metric corresponds to a low-level program feature, which defines particular application characteristics, and is available as a user-controllable knob. The user can choose to fix the values of some (or all) core metrics to generate targeted program behavior. For the remaining set of core metrics (if any) whose values are not fixed by the user, Genesys randomizes their values within reasonable bounds in order to achieve well-rounded program state-space coverage around the target behavior. By allowing each workload-specific metric to be controlled using easy-to-use programmable knobs, Genesys allows the creation of targeted benchmarks with desired program features. Together, the values chosen for the core metrics act as unique profiles for the synthetic workloads. These workload profiles are fed into a code generator algorithm that uses the target metric values to generate a suite of synthetic applications. Together, these applications form a set of unique programs, which target particular aspects of the program behavior depending upon the choice of the core metrics.

In this section, the core workload-specific metrics used by Genesys are first discussed, followed by the workload generation methodology. The core feature

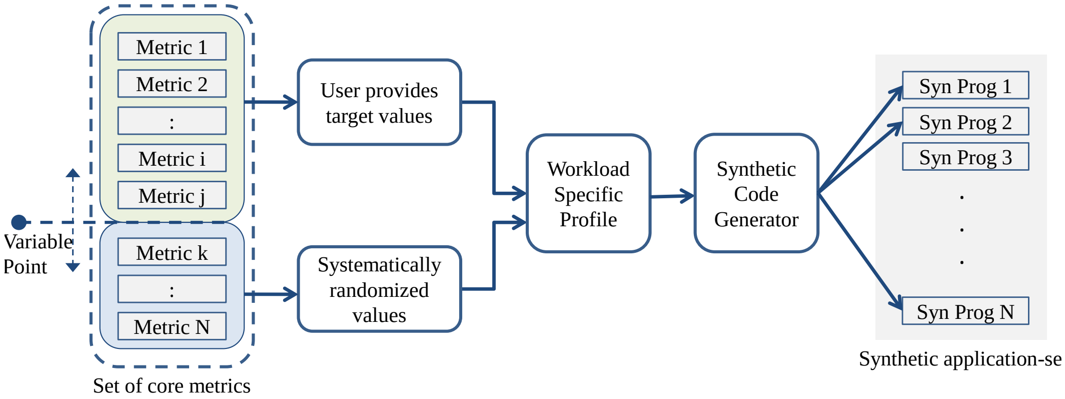


Figure 8.1: Genesys's Overall Methodology and Framework

set is divided at a high-level into three categories, depending upon the aspects of program behavior that the individual metrics control. The three categories, together with their associated sub-categories and component metrics are shown in Table 8.1 and are described below. It should be noted that the set of core metrics used in this paper are not meant to be conclusive, rather, they are key metrics that affect different aspects of program behavior. Nevertheless, Genesys's framework is flexible enough to add new metrics to control other aspects of program behavior.

8.1.1 Instruction-level Characteristics

These metrics correspond to the instruction-level behavior of the applications.

a. Instruction mix: The first metric is the application's instruction mix (IMIX). Genesys categorizes instructions into fractions of loads and stores (memory), control-flow, integer and floating-point instructions. It should be noted that the framework is very flexible and can be easily extended to support specific instruction categories. The target IMIX can be provided as an input to Genesys directly, in which case it generates programs having the desired overall IMIX. Otherwise, IMIX fractions, randomized within bounded ranges, are used to generate the suite of synthetic applications.

b. Instruction count: The second metric that is considered is instruction count (IC), which controls the static instruction footprint of the application. IC can be provided by the user directly or estimated automatically based on the target instruction cache miss rate (ICMR, metric 3). If the ICMR metric is provided, Genesys determines the number of static instructions to instantiate in the synthetic benchmark to achieve the target ICMR. An initial estimate of the number of static instructions is made based on the assumption of a default instruction cache (Icache) size/configuration. This serves as an initial estimate only, the final static code size is further tuned to achieve the target ICMR.

c. Instruction-level parallelism: Instruction-level parallelism (ILP) is an important determinant of an application's performance. Tight producer-consumer chains in program sequences limit ILP and performance because of dependency-induced serialization effects. Genesys models ILP by controlling the dependencies between instructions in the application sequence using the dependency distance metric. Dependency distance is defined as the total number of instructions in the dynamic instruction stream between the production (write) and consumption (read) of a register/memory location. We classify dependency distance into 32 bins (values varying between 1 to 32 and higher), where each bin represents the fraction of in-

Table 8.1: Genesys' Workload Metrics

Category	Metrics	Count
Instruction-level Characteristics	1. Instruction mix	5 categories
	2. Instruction count	1
	3. Instruction cache miss rate (ICMR)	1
	4. Instruction level parallelism (ILP)	32 bins
Control-flow Characteristics	5. Average basic block size	1
	6. Branch transition rate (BTR)	1
	7. Branch misprediction rate	1
Memory-access Characteristics	8. Data footprint	1
	9. Regular/irregular behavior	1
	10. Spatial locality stride bins	32 bins
	11. Temporal locality bins	8 bins
	12. L1/L2 Data cache miss rates	2

structions having that particular dependency distance. The desired dependency distance can be provided as an input to Genesys or automatically randomized (within bounds) to generate synthetic programs with varying degrees of ILP.

8.1.2 Control-flow Characteristics

These metrics affect the program's control-flow behavior.

a. Average basic block size: Average basic block size is an important metric because it determines the average number of instructions that can be executed in the program sequence without executing any control instructions. This can affect performance significantly depending on the branch predictor performance. Again, this metric could be provided directly as an input or inferred from the ICMR metric (described before) and the fraction of control instructions.

b. Branch predictability model: Genesys considers two other control-flow metrics: branch transition rate (BTR) and branch misprediction rate. Prior research studies [34] have shown that an application's branch misprediction rate is highly correlated with the transition characteristics of the branch instructions. The key idea behind this correlation is that, the higher the transition probability of a branch instruction, the more difficult it is to predict its next direction and vice versa. To model a certain BTR, a set of control instructions are chosen to be modeled with high transition probabilities (frequent switching) and the remaining branch instructions are are modeled to have very low transition probabilities (infrequent switching activity). Similarly, Genesys can also model the transition probability of individual branch instructions in a directly correlated fashion to achieve a target branch misprediction rate (metric 7).

8.1.3 Memory-level Characteristics

This section describes metrics that affect the memory performance (data-side) of applications.

a. Data footprint: The data footprint metric determines the range of data addresses accessed by the synthetic application during its execution time. This is important because it can determine performance of different levels of caches and memory based on how large the footprint is with respect to the available cache size and memory structure. It controls the size of the memory regions that are accessed by the synthetic application.

b. Memory access regularity: This metric determines if the memory accesses made by load/store instructions of an application should have regular or irregular behavior. For irregular memory behavior, Genesys generates load/store instructions that access allocated and initialized memory regions based on a randomly generated sequence. Regular memory behavior is achieved using additional metrics (spatial-temporal locality or L1/L2 cache miss rate metrics) as described below.

c. Spatial and temporal locality: The principle of data locality and its impact on application's performance is widely recognized. Genesys models regular data memory accesses using simple strided stream classes over fixed-size data arrays, where strides are defined to be the difference between consecutive effective addresses. Strides can be provided directly as an input to Genesys to control spatial locality characteristics (bins representing strides from -1K to 1K in multiples of 64B). Genesys also provides knobs to control the temporal locality (8 bins expressed as powers-of-two from 0 to 128) in the memory accesses. The temporal locality metric controls the number of unique memory accesses between access to the same memory location and affects the achieved cache miss rates as well. Together, the stride and temporal locality bin values are used to generate the sequence of memory addresses.

Genesys can also automatically estimate the strides (offsets) of the load/store instructions based on the target data cache miss rate statistics. This approach is similar to that adopted by Bell et al. [11]. The strides for a particular memory access are determined first by matching the L1 hit rate of a load/store, followed by the L2 hit rate. We generate a table that holds the correlation between L1/L2 cache hit rates and the corresponding stride values to be used. We use the target L1/L2 hit rate information along with this table to generate stride values of load and store instructions. By treating all memory accesses as streams and working from a base cache configuration, the memory access model is kept simple.

8.1.4 Genesys's Workload Generation Methodology

The workload synthesis algorithm, based on the metrics discussed previously, is as follows:

1. Generate a random number in the interval [0, 1] and select a basic block based on this number and the block's access frequency.

2. The basic block's size is calculated in order to satisfy the mean and standard deviation of the target basic block size.

3. The basic block is populated with instructions based on the IMIX metrics, while ensuring that the last instruction of the basic block is a branch instruction.

4. Every instruction is assigned a dependency distance (i.e., a previous instruction that it is dependent upon) in order to satisfy the dependency distance criterion.

5. The load and store instructions are assigned a stride-offset based on the memory access model described in the previous section (regular or irregular).

6. An X86 test operation is used to set the condition codes that affect the outcome of the conditional branch instruction at the end of each basic block. The

"test" operand is controlled to achieve the target BTR metric.

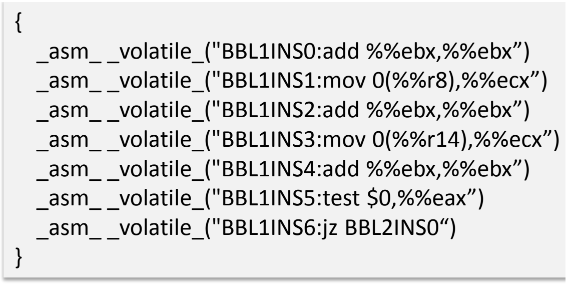
7. The number of generated basic blocks is incremented.

8. If the target number of basic blocks have been generated, go to step 9, else update the individual metric distributions and go back to step 1.

9. Available architected registers are assigned to each instruction while satisfying the data dependencies established in step 4.

10. The above instruction sequence is generated as a part of two-level nested loops where the inner loop controls the application's data footprint and the outer loop controls the number of dynamic instructions (overall runtime). Every static load or store instruction resets to the first element of the strided memory streams and re-walks the entire stream in the outer loop iterations.

The code generator generates the instruction sequence using C-language with embedded X86-based assembly instructions. An example code snippet is shown in Figure 8.2. The code generator can be modified to generate instructions for a different ISA. The code is encompassed inside a main header and malloc library calls are used to allocate memory for the data streams. Volatile directives are used for each asm statement and the program is compiled using the lowest compiler optimization level (-O0 with gcc) in order to prevent the compiler from optimizing out the machine instructions.



```
{
   _asm_ _volatile_("BBL1INS0:add %%ebx,%%ebx”)
   _asm_ _volatile_("BBL1INS1:mov 0(%%r8),%%ecx”)
   _asm_ _volatile_("BBL1INS2:add %%ebx,%%ebx”)
   _asm_ _volatile_("BBL1INS3:mov 0(%%r14),%%ecx”)
   _asm_ _volatile_("BBL1INS4:add %%ebx,%%ebx”)
   _asm_ _volatile_("BBL1INS5:test $0,%%eax”)
   _asm_ _volatile_("BBL1INS6:jz BBL2INS0“)
}
```

Figure 8.2: Example Synthetic Code Snippet

8.2 Evaluation

This section describes the experimental setup and results in detail.

8.2.1 Experimental Setup

All the experiments are conducted on Intel Xeon E5-2430 v2 server class machines with Ivy-bridge micro-architecture based processing cores, three levels of caches (1.5MB L2 and 15MB L3 cache) and 64 GB of main memory. The Linux perf tool [53] is used for measuring the hardware performance of different applications. Power consumption is monitored using Intel's RAPL counters.

To show the efficacy of Genesys, the synthetic programs generated using Genesys are compared with a program set comprised of benchmarks drawn from several popular benchmarking suites (hereafter referred to as the REAL program set). The REAL program set includes 70 standard benchmarks: 29 benchmarks from the SPEC CPU2006 suite (using ref inputs), 20 benchmarks from MiBench, 10 benchmarks from MediaBench and 11 TPC-H queries. Details about the synthetic programs created using Genesys (hereafter referred to as the GEN program sets) are provided in the following sections. Each GEN program's size is restricted so that each program can complete within 1 to 15 seconds on the target machine.

8.2.2 Results: State-space Coverage

This section shows how Genesys can be leveraged to automatically create programs with different features leading to a wider coverage of the program state-space. To do so, the program state-space coverage provided by the REAL program set is compared against the GEN program set. For this study, the GEN set consists of 500 synthetic programs created using Genesys. The GEN programs are uniquely generated by using random combinations of individual metric values (chosen systematically within respective metric bounds). It takes roughly a few (∼5-20) seconds to generate each GEN program and every program completes execution within 1 to 15 seconds on the target machine. Thus, the total run-time of all the GEN programs is roughly equal to the total run-time of the 70 programs from the REAL program set due to the significantly longer run-times of several REAL benchmarks.

In order to compare the program state-space coverage achieved by either program sets, a novel metric (***SpreadRatio***) is defined, which is defined as the ratio of the area of the convex hull envelope of the REAL versus GEN program features. The convex hull [9] of a set S of points in the Euclidean space is defined as the smallest convex set that contains S. The convex hull of a set of points S in n dimensions is the intersection of all convex sets containing S. For N points $p_1, ..., p_N$ in n-dimensions, the convex hull C is given by the expression:

$$C \equiv \sum_{j=1}^{N} \lambda_j p_j : \lambda_j \geq 0 \, \forall j \; and \sum_{j=1}^{N} \lambda_j = 1$$

Based on this definition of a convex hull, let C_{REAL} represent the convex hull of the points covered by the REAL program set and C_{GEN} represent the convex hull of the points covered by the GEN program set. Then, *SpreadRatio* can be defined using the following expression:

$$SpreadRatio = \frac{Area(C_{GEN})}{Area(C_{REAL})}$$

Next, the state-space coverage of the GEN and REAL programs is compared using the *SpreadRatio* metric. To better demonstrate the degree of controllability provided by Genesys, the GEN and REAL programs are first compared using subsets of performance characteristics, followed by using the entire set. Since the number of metrics is large, it is difficult to visualize all the variables simultaneously to draw any meaningful conclusions. Thus, statistical data analysis techniques are used to simplify the comparison. Using a large number of correlated variables tends to unduly overemphasize the importance of a particular property. Therefore, raw data are first normalized to a unit normal distribution (mean = 0, standard deviation = 1) and then pre-processed using Principal Component Analysis (PCA) [21]. PCA is an effective statistical data analysis technique to reduce the dimensionality of a data-set, while maintaining most of its original information. PCA transforms the original variables into a set of uncorrelated “principal components” (PC’s). If significant correlation exists between the original variables, then most of the original information will be captured using just the top few PC’s.

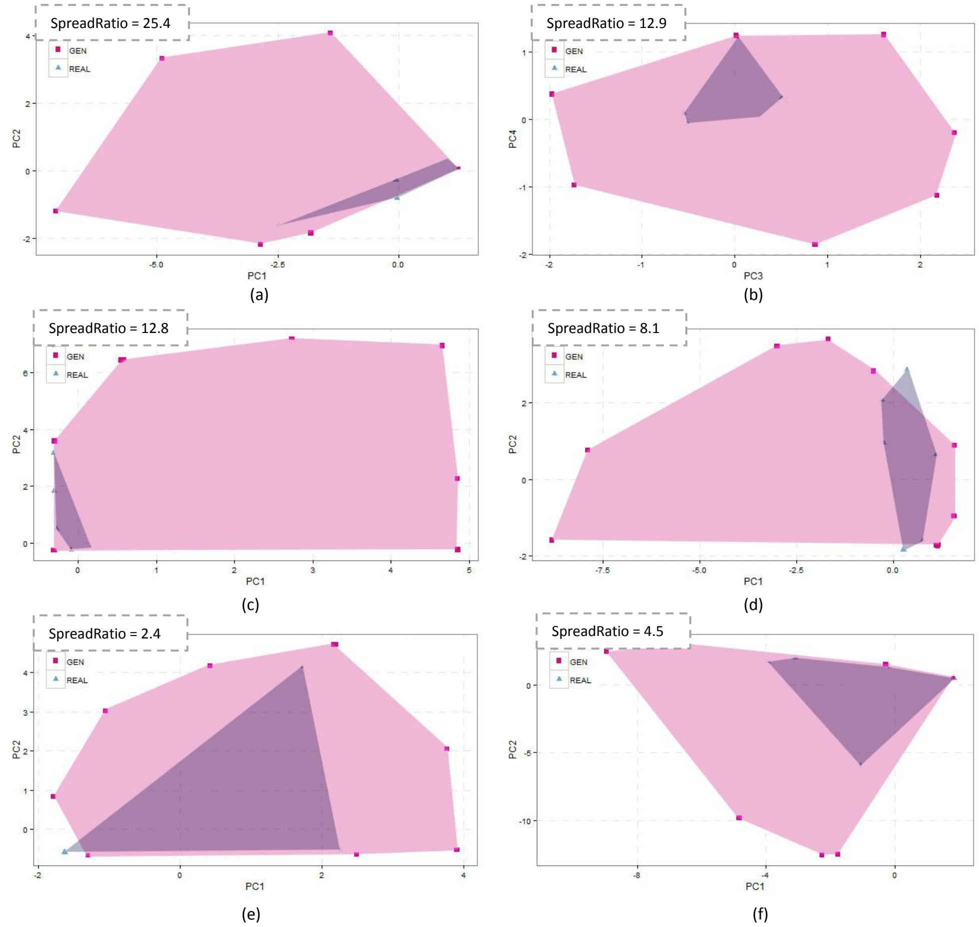


Figure 8.3: State-space Coverage of REAL and GEN Programs using (a) Cache/memory Behavior - PC1 vs PC2 (b) Cache/memory Behavior - PC3 vs PC4 (c) TLB Behavior (d) Instruction-level Behavior (e) Control-flow Behavior (f) Overall Characteristics

The first set of experiments compare the memory subsystem performance behavior of the GEN and REAL program sets based on the L1 Dcache, Icache, L2 and LLC misses per kilo instruction (MPKI) metrics. Figures 8.3a and 8.3b show the scatterplots of the top 4 principal components (PC1 vs PC2 and PC3 vs PC4), respectively. It can be observed that applications from the REAL set do not stress the instruction side performance much, and as a result, the Icache MPKI for the REAL programs is mostly very low. Such behavior is different from the emerging big-data and cloud applications' behaviors, which have been shown to stress the instruction side performance more heavily [67, 70]. Nevertheless, by controlling Genesys's I/D memory-side metrics, it is possible to create programs that stress the instruction and data-side performance to varying degrees. Overall, the GEN programs provide 25.4 times (SpreadRatio = 25.4) higher coverage area than the REAL programs for the first two principal components and 12.9 times (SpreadRatio = 12.9) higher coverage than REAL programs in the PC3 versus PC4 space.

Figure 8.3c compares the I/D TLB performance of the REAL and GEN programs. The x-axis corresponds to the ITLB MPKI whereas the y-axis corresponds to the DTLB MPKI of the programs. Although the standard benchmarks provide good coverage in terms of DTLB performance, but none of the REAL programs stress the ITLB much, whereas the GEN programs provide extensive state-space coverage in terms of both the instruction and data TLB performance. Overall, the GEN program set provides 12.8 times higher coverage than the REAL program set (SpreadRatio = 12.8).

The next set of experiments focuses on the instruction-level performance characteristics (shown in Figure 8.3d), based on the overall IPC, μOps/instruction, IMIX and ILP (given by dependency-driven pipeline stalls) metrics. Again, the GEN programs provide 8.1 times broader state-space coverage as compared to the

Table 8.2: Hardware Performance Features Used to Compare REAL and GEN Programs.

Performance Features	
μOps/instruction	FP Ops/instruction
branch/instruction	branch miss/instruction
Icache MPKI	Dcache MPKI
ITLB MPKI	DTLB MPKI
L2 MPKI	LLC MPKI

REAL programs for the instruction-level metrics.

The control-flow performance coverage of the REAL and GEN programs is shown in Figure 8.3e. For this experiment, control-flow performance corresponds to three metrics - the branch misprediction rate, average basic block size and percentage of branch instructions. It can be observed that the REAL programs have much better branch performance coverage as compared to their cache and TLB performance, but the GEN programs still outperform the REAL set by providing 2.4x higher coverage (SpreadRatio = 2.4).

Figure 8.3f shows the state-space coverage provided by the REAL and GEN programs in the PC1 versus PC2 space using all performance features shown in Table 8.2 including IPC. Overall, GEN provides 4.5x higher state-space coverage as compared to the REAL set using all the program features. It can be concluded that Genesys's methodology of controlling key low-level application metrics allows it to easily generate programs with varied performance characteristics.

8.3 Summary

This chapter presented Genesys, a novel workload generation framework that enables the systematic generation of synthetic applications, providing a wider coverage of program behavior state-space. Genesys allows the user to control a set of key workload-specific characteristics using easy-to-use, programmable knobs. Thus, Genesys enables generating synthetic applications targeting specific program properties. In order to compare the state-space coverage provided by different sets of applications, this chapter defined a novel metric called SpreadRatio that is based on the area of the convex hull envelope surrounding the program points. It is demonstrated that by using automatically generated program sets, it is possible to achieve over 11 times higher state-space coverage than that provided by popular, standard benchmarks such as SPEC CPU2006, MiBench, MediaBench and TPC-H.

Chapter 9

Conclusion and Future Work

Fast and accurate design-space exploration is a critical requirement for enabling future hardware designs. Early computer design evaluation is performed using detailed performance models such as execution-driven simulators or RTL-based models. Although accurate, such detailed performance modeling techniques suffer from several challenges. First, several emerging big-data applications are often complex targets to evaluate on early performance models as running similar applications requires handling their complex software layers, back-end databases and third-party libraries, which are challenging (often impossible) to support on most early performance models. Second, detailed performance models are significantly slower than real hardware, which makes it difficult to analyze the complete execution characteristics of these long-running applications. Finally, effective modeling techniques require access to either the application source code or traces. Unfortunately, source code or exact traces of end-user workloads are often inaccessible due to their proprietary nature. Thus, computer designers and researchers often find it difficult to create optimal designs targeting end-user applications.

This dissertation focuses on developing techniques, which help computer designers gain a better understanding of end-user workloads and improve the speed and efficiency of early performance evaluation of emerging applications and architectures. These techniques rely on proxy benchmarking, i.e., replicating the performance behavior of end-user applications using miniaturized synthetic proxy benchmarks. These benchmarks then can be used for early computer design space exploration without compromising the privileged nature of software/data, while significantly reducing the simulation times. The following section summarizes the key contributions made in this dissertation.

9.1 Summary

This work describes five techniques to improve proxy benchmark generation for emerging workloads and architectures.

Poor memory system performance is a critical overall performance bottleneck for several applications. Designing optimal memory systems for improved performance and energy efficiency requires computer architects to have a deep understanding of the memory access behavior of the end-user workloads. To facilitate fast and efficient evaluation of futuristic memory hierarchies, this dissertation proposes HALO, a hierarchical memory access locality modeling technique that can statistically capture the spatial and temporal locality of applications, while incurring less meta-data storage overhead. HALO discovers patterns by decomposing an application's memory accesses into a set of independent streams that are constrained to a smaller region of memory and capturing fine-grained access patterns within localized regions using repeating stride transitions. This allows the representation of complex workloads through the composition of a set of smaller and simpler building blocks. Moreover, different programs have different locality behavior. HALO exploits this observation to achieve higher meta-data storage efficiency by capturing multi-level stride transitions, which are tailored to an application's locality patterns. However, modeling locality within localized streams alone is not sufficient to recreate the original application's memory behavior. HALO also models how accesses to the individual localized streams are interleaved with respect to each other by leveraging coarse-grained temporal locality tracking. HALO proxies achieve over $\sim$96% accuracy in replicating performance of several emerging applications across different cache, prefetcher, TLB and DRAM memory configurations, while outperforming state-of-the-art WEST and STM techniques.

Next, this dissertation extends the workload cloning approach GPUs. In the last few years, GPUs have emerged as a highly popular computation platform for applications beyond graphics. Programmers exploit these massively parallel architectures in diverse domains (e.g., linear algebra, bio-informatics, high performance computing, etc.). To enable fast and efficient memory system exploration in GPUs, this dissertation proposes, G-MAP, a novel proxy benchmark generation methodology and framework that statistically models the patterns in GPU application's memory access streams. G-MAP models the regularity in code-localized memory access patterns of GPU applications and parallelism in the GPU's execution model to create miniaturized memory traces or proxies. G-MAP proxies achieve over 90% accuracy in replicating cache and memory performance of original GPU applications across thousands of cache, prefetcher, memory design configurations, while significantly reducing the simulation time and storage requirements.

The next proposal is a system-level proxy benchmarking methodology, CAMP for creating miniature proxy benchmarks that model both core performance and memory locality of big-data applications. CAMP proxies are representative of the performance of real-world big-data applications and yet converge to results quickly and without any complex software-stack support. To create the system-level proxy benchmarks, this dissertation analyzes the key drivers of big-data application performance (e.g., larger code/data memory footprints, operating system, and other run-time effects). Such effects are often not highly significant in traditional desktop/general-purpose applications and thus, are not modeled by prior workload cloning proposals. To model the core performance, CAMP improves upon existing methods for generating proxy instruction streams by capturing and modeling the instruction-level parallelism, instruction types, control-flow behavior, etc. CAMP adds an improved memory locality profiling approach that captures both the spatial and temporal locality of applications. Furthermore, CAMP leverages a novel proxy generation and modeling/replay methodology that integrates the core and memory locality models to create accurate system-level proxy benchmarks.

The next proposal, PerfProx is a system-level proxy benchmark generation methodology that captures the key performance metrics affecting big-data application performance using hardware performance counters and stochastically models them to create miniaturized proxy benchmarks. The proposed performance counter based application characterization and associated extrapolation into generic parameters for proxy code generation enables fast analysis (runs almost at native hardware speeds, unlike prior workload cloning proposals) and proxy generation of complex, long-running big-data applications with reasonable accuracy. The generated proxy benchmarks replicate performance behavior of real-world cloud applications without needing any back-end database/complex software stack support and thus, help to overcome the challenges in benchmarking such workloads on early performance models.

Finally, Genesys allows creating futuristic workloads with performance behavior that does not currently exist by systematically tweaking the program characteristics, used as an input to the proxy generation framework. Genesys proxies can cover anticipated code trends and can represent futuristic workloads before the workloads even exist.

9.2 Future Work

While this dissertation makes significant contributions to improve proxy benchmark generation methodologies, there are still opportunities for future work. This section list possible future work.

While the proposed system-level proxy benchmark generation schemes model aggregate performance of the end-user workloads, they do not capture dynamic phase-level behavior of the workloads. Modeling fine-grain phase level behavior can help to improve accuracy and fidelity of the generated proxies. The proposed schmese can be extended to incorporate fine-grain phase behavior into the proxies.

The current proxy benchmarks capture the performance behavior of the target end-user applications, including effects such as data footprint, query type, configuration, input data-set, etc. If any of these parameters change in the original application, a new proxy has to be generated accounting for the change in parameters or configuration. It will be useful if such scenarios can be analyzed and the current proxy generation methodology can be extended to yield scalable proxies or proxies with programmable knobs. For example, with the proliferation of cloud-based computing systems, applications are scaling to larger configurations. Different applications exhibit different scaling behavior with different hardware and software configurations. The application scaling behavior can be investigated and these effects be incorporated to create scalable proxy applications.

Bibliography

[1] GraphChi. `https://github.com/GraphChi/graphchi-cpp`.

[2] GraphLab. `www.graphlab.org`.

[3] Memcached. `www.memcached.org`.

[4] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[5] Amro Awad and Yan Solihin. STM: Cloning the spatial and temporal memory access behavior. In *HPCA*, pages 237–247, 2014.

[6] Ali Bakhoda et al. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*, pages 163–174. IEEE Computer Society, 2009.

[7] Ganesh Balakrishnan and Yan Solihin. WEST: Cloning data cache behavior using stochastic traces. In *HPCA*, pages 387–398. IEEE Computer Society, 2012.

[8] Ganesh Balakrishnan and Yan Solihin. MEMST: Cloning memory behavior using stochastic traces. In *MEMSYS*, pages 146–157, 2015.

[9] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions On Mathematical Software*, 22(4):469–483, 1996.

[10] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. *SIGARCH Comput. Archit. News*, 26(3), April 1998.

[11] Robert H. Bell, Jr. and Lizy K. John. Improved automatic testcase synthesis for performance model validation. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 111–120, 2005.

[12] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, July 1975.

[13] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[14] Alexandru Boicea, Florin Radulescu, and Laura Ioana Agapin. MongoDB vs oracle – database comparison. In *EIDWT*, pages 330–335. IEEE Computer Society, 2012.

[15] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring

the level of abstraction for scalable & accurate parallel multi-core simulations. In *SC*, 2011.

[16] Cassandra. `wiki.apache.org/cassandra/FrontPage`.

[17] Shuai Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, 2009.

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[19] Etem Deniz and Alper Sen. MINIME-GPU: Multicore benchmark synthesizer for gpus. *ACM Trans. Archit. Code Optim.*, 12(4):34:1–34:25, November 2015.

[20] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. *SIGPLAN Not.*, 38(5):245–257, May 2003.

[21] G. Dunteman. *Principal Component Analysis*. Sage Publications, 1989.

[22] L. Eeckhout, K. de Bosschere, and H. Neefs. Performance analysis through synthetic trace generation. In *ISPASS*, pages 1–6, 2000.

[23] Lieven Eeckhout, Robert H. Bell Jr., Bastiaan Stougie, Koen De Bosschere, and Lizy Kurian John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *ISCA*, 2004.

[24] Lieven Eeckhout, Hans Vandierendonck, and Koen De Bosschere. Workload design: Selecting representative program-input pairs. In *PACT*, volume 0, page 83. IEEE Computer Society, 2002.

[25] L. Van Ertvelde and L. Eeckhout. Benchmark synthesis for architecture and compiler exploration. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–11, Dec 2010.

[26] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. *SIGPLAN Not.*, 47(4):37–48, March 2012.

[27] Avrilia Floratou, Nikhil Teletia, David J. DeWitt, Jignesh M. Patel, and Donghui Zhang. Can the elephants handle the nosql onslaught? *Proc. VLDB Endow.*, 5(12):1712–1723, August 2012.

[28] Karthik Ganesan, Jungho Jo, W. Lloyd Bircher, Dimitris Kaseridis, Zhibin Yu, and Lizy K. John. System-level max power (SYMPO): A systematic approach for escalating system-level power consumption using synthetic benchmarks. In *PACT*, pages 19–28, 2010.

[29] Karthik Ganesan, Jungho Jo, and Lizy K. John. Synthesizing memory-level parallelism aware miniature clones for spec cpu2006 and implantbench workloads. In *ISPASS*, 2010.

[30] Karthik Ganesan and Lizy Kurian John. Automatic generation of miniaturized synthetic proxies for target applications to efficiently design multicore processors. *IEEE Trans. Computers*, 63(4):833–846, 2014.

[31] Wanling Gao, Yuqing Zhu, Zhen Jia, Chunjie Luo, Lei Wang, Zhiguo Li, Jianfeng Zhan, Yong Qi, Yongqiang He, Shimin Gong, Xiaona Li, Shujie Zhang, and Bizhu Qiu. BigDataBench: a big data benchmark suite from web search engines. *CoRR*, abs/1307.0320, 2013.

[32] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC-4.*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[33] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and more flexible program analysis. *Journal of Instruction Level Parallelism*, 2005.

[34] Michael Haungs, Phil Sallee, and Matthew K. Farrens. Branch transition rate: A new metric for improved branch classification analysis. In *HPCA*, pages 241–250. IEEE Computer Society, 2000.

[35] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, June 2009.

[36] HP Linpack. `http://icl.eecs.utk.edu/hpl/`.

[37] Maria Indrawan-Santiago. Database research: Are we at a crossroad? reflection on nosql. In *NBIS*, pages 45–51. IEEE Computer Society, 2012.

[38] Aamer Jaleel, Robert S. Cohn, Chi keung Luk, and Bruce Jacob. CMP$im: A pin-based on-the-fly multi-core cache simulator. In *Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2008.

[39] Zhen Jia, Lei Wang, Jianfeng Zhan, Lixin Zhang, and Chunjie Luo. Characterizing data analysis workloads in data centers. *CoRR*, abs/1307.8013, 2013.

[40] Zhanpeng Jin and Allen C. Cheng. Implantbench: Characterizing and projecting representative benchmarks for emerging bioimplantable computing. *IEEE Micro*, 28, 2008.

[41] Russ Joseph and Margaret Martonosi. Run-time power estimation in high performance microprocessors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, ISLPED '01, pages 135–140, 2001.

[42] A. Joshi, L. Eeckhout, R. H. Bell, and L. John. Performance cloning: A technique for disseminating proprietary applications as benchmarks. In *IEEE IISWC*, pages 105–115, Oct 2006.

[43] Ajay Joshi, Lieven Eeckhout, and Lizy John. The return of synthetic benchmarks. In *Proceedings of the 2008 SPEC Benchmark Workshop*, pages 1–11, San Francisco, CA, USA, 1 2008.

[44] Ajay Joshi, Aashish Phansalkar, L. Eeckhout, and L. K. John. Measuring benchmark similarity using inherent program characteristics. *IEEE Transactions on Computers*, 55(6):769–782, 2006.

[45] David Kadjo, Jinchun Kim, Prabal Sharma, Reena Panda, Paul Gratz, and Daniel Jimenez. B-fetch: Branch prediction directed prefetching for chip-multiprocessors. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 623–634, 2014.

[46] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. Performance characterization of a quad pentium pro SMP using OLTP workloads. In *ISCA*, pages 15–26, Washington, DC, USA, 1998.

[47] H Kim, J Lee, N. B Lakshminarayana, J Sim, J Lim, and T Pho. MacSim: A cpu-gpu heterogeneous simulation. *Framework User Guide, Georgia Institute of Technology*, 2012.

[48] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible DRAM simulator. *IEEE Computer Architecture Letters*, 2015.

[49] Sanjeev Kumar and Christopher Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, pages 357–368, Washington, DC, USA, 1998. IEEE Computer Society.

[50] Michael LeBeane, Jee Ho Ryoo, Reena Panda, and Lizy Kurian John. Watt Watcher: Fine-grained power estimation for emerging workloads. In *27th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2015, Florianópolis, Brazil, October 17-21, 2015*, pages 106–113, 2015.

[51] Jaekyu Lee, Nagesh B. Lakshminarayana, Hyesoon Kim, and Richard Vuduc. Many-thread aware prefetching mechanisms for GPGPU applications. *MICRO*, pages 213–224, 2010.

[52] Shin Ying Lee and Carole Jean Wu. Characterizing the latency hiding ability of gpus. In *ISPASS*, pages 145–146, 2014.

[53] Linux perf tool. `https://perf.wiki.kernel.org/index.php/Main_Page`.

[54] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[55] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970.

[56] MongoDB. `http://mongodb.org`.

[57] MongoDB architecture guide. `https://www.mongodb.com/collateral/mongodb-architecture-guide`.

[58] MySQL. `http://www.mysql.com`.

[59] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *HPCA*, 2004.

[60] NVIDIA CUDA c/c++ sdk code samples, 2011.

[61] Cedric Nugteren, Gert-Jan van den Braak, and Henk Corporaal. A detailed GPU cache model based on reuse distance theory. In *HPCA*, pages 37–48, Los Alamitos, CA, USA, 2014. IEEE Computer Society.

[62] NVIDIA. CUDA C programming guide 5.5, 2013.

[63] NVIDIA's next generation CUDA compute architecture, Fermi, 2009.

[64] Frank Olken. Efficient methods for calculating the success function of fixed space replacement policies. *Report No. LBL-12370, Berkeley, CA : Lawrence Berkeley Nat. Lab.*, 3:153–154, 1981.

[65] Mark Oskin, Frederic T. Chong, and Matthew Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor designs. In *ISCA*, ISCA '00, pages 71–82, 2000.

[66] Reena Panda, Yasuko Eckert, Nuwan Jayasena, Onur Kayiran, Michael Boyer, and Lizy Kurian John. Prefetching techniques for near-memory throughput processors. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 40:1–40:14, 2016.

[67] Reena Panda, Christopher Erb, Michael Lebeane, Jeeho Ryoo, and Lizy Kurian John. Performance characterization of modern databases on out-of-order cpus. In *IEEE SBAC-PAD*, 2015.

[68] Reena Panda, Daniel A. Jimenez, and Paul V. Gratz. B-fetch: Branch prediction directed prefetching for in-order processors. *IEEE Computer Architecture Letters*, 11:41–44, 2012.

[69] Reena Panda and Lizy John. Proxy benchmarks for emerging big-data workloads. In *The 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.

[70] Reena Panda and Lizy Kurian John. Data analytics workloads: Characterization and similarity analysis. In *IPCCC*, pages 1–9. IEEE, 2014.

[71] Reena Panda and Lizy Kurian John. Proxy benchmarks for emerging big-data workloads. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2017.

[72] Reena Panda, Shuang Song, Joseph Dean, and Lizy Kurian John. Wait of a decade: Did SPEC CPU2017 benchmarks broaden the performance spectrum? In *HPCA*, 2018.

[73] Reena Panda, Xinnian Zheng, Andreas Gerstlauer, and Lizy Kurian John. CAMP: Accurate modeling of core and memory locality for proxy generation of big-data applications. In *DATE*, 2018.

[74] Reena Panda, Xinnian Zheng, and Lizy John. Accurate address streams for LLC and beyond (SLAB): A methodology to enable system exploration. In *IEEE ISPASS*, 2017.

[75] Reena Panda, Xinnian Zheng, Shuang Song, Jee Ho Ryoo, Michael LeBeane, Andreas Gerstlauer, and Lizy Kurian John. Genesys: Automatically generating representative training sets for predictive benchmarking. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS*, pages 116–123, 2016.

[76] Reena Panda, Xinnian Zheng, Jiajun Wang, Andreas Gerstlauer, and Lizy John. Statistical pattern based modeling of GPU memory access streams. In *ACM Design Automation Conference (DAC)*, 2017.

[77] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSSx86: A Full System Simulator for x86 CPUs. In *Design Automation Conference (DAC)*, 2011.

[78] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large intel®itanium®programs with dynamic instrumentation. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 81–92, 2004.

[79] A. Phansalkar, A. Joshi, L. Eeckhout, and L.K. John. Measuring program similarity: Experiments with spec cpu benchmark suites. In *ISPASS*, volume 0, pages 10–20, 2005.

[80] Jason Power, Joel Hestness, Marc S. Orr, Mark D. Hill, and David A. Wood. gem5-gpu: A heterogeneous CPU-GPU simulator. *IEEE CAL*, 14(1):34–36, Jan 2015.

[81] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: A cycle accurate memory system simulator. *IEEE Comput. Archit. Lett.*, 10(1):16–19, January 2011.

[82] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. *SIGOPS Oper. Syst. Rev.*, 32(5):115–126, October 1998.

[83] A. D. Samples. Mache: No-loss trace compaction. In *Proceedings of the 1989 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS ’89, pages 89–97, 1989.

[84] André Seznec. A new case for the TAGE branch predictor. *IEEE MICRO*, 2011.

[85] Minglong Shao, Anastassia Ailamaki, and Babak Falsafi. DBmbench: Fast and accurate database workload representation on modern microarchitecture. In *CASCON*, pages 254–267, 2005.

[86] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, 2001.

[87] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5):45–57, October 2002.

[88] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H. Pugsley, and Zeshan Chishti. Efficiently prefetching complex address patterns. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, 2015.

[89] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In *PPoPP*, 2012.

[90] Marc Snir and J. Yu. On the theory of spatial and temporal locality. *Technical Report UIUCDCS-R-2005-2611*, 2005.

[91] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 252–263, Washington, DC, USA, 2006. IEEE Computer Society.

[92] SPEC CPU2000. `https://www.spec.org/cpu2000`.

[93] SPEC CPU 2006 Benchmarks. `www.spec.org/cpu2006`.

[94] SPEC CPU 2017 Benchmarks. `www.spec.org/cpu2017`.

[95] SPECjbb 2005. `https://www.spec.org/jbb2005/`.

[96] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chan, Nasser Anssari, Geng Daniel Liu, and Wen-mei W. Hwu. IMPACT technical report. *IMPACT-12-01, University of Illinois at Urbana-Champaign*, March 2012.

[97] Tao Tang et al. Cache miss analysis for gpu programs based on stack distance profile. In *ICDCS*, pages 623–634, 2011.

[98] D. Thiebaut, J. L. Wolf, and H. S. Stone. Synthetic traces for trace-driven simulation of cache memories. *IEEE Transactions on Computers*, 41(4):388–410, Apr 1992.

[99] TPC-C Benchmark Suite. `http://www.tpc.org/tpcc`.

[100] TPC-H Benchmark Suite. `http://www.tpc.org/tpch`.

[101] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: A survey. *ACM Comput. Surv.*, 29, 1997.

[102] Luk Van Ertvelde and Lieven Eeckhout. Dispersing proprietary applications as benchmarks through code mutation. *ASPLOS*, pages 201–210, 3 2008.

[103] Hans Vandierendonck and Pedro Trancoso. Building and validating a reduced TPC-H benchmark. In *14th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 383–392. IEEE, 2006.

[104] Jiajun Wang, Reena Panda, and Lizy John. Prefetching for cloud workloads: An analysis based on address patterns. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2017.

[105] Jiajun Wang, Reena Panda, and Lizy John. SelSMaP: A selective stride masking prefetching scheme. In *The 35th IEEE International Conference on Computer Design*, 2017.

[106] Yipeng Wang, Ganesh Balakrishnan, and Yan Solihin. MeToo: Stochastic modeling of memory traffic timing behavior. In *PACT*, pages 457–467, 2015.

[107] Jonathan Weinberg and Allan Edward Snavely. Accurate memory signatures and synthetic address traces for HPC applications. In *ICS*. ACM, 2008.

[108] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 84–97, New York, NY, USA, 2003. ACM.

[109] Z. Yu et al. GPGPU-MiniBench: Accelerating gpgpu micro-architecture simulation. *IEEE Transactions on Computers*, 64(11):3153–3166, Nov 2015.

Vita

Reena Panda was born in Balasore, India. She did her schooling in Kendriya Vidyalaya No.1, Bhubaneswar India. She received her bachelor's degree in electrical engineering from the National Institute of Technology (NIT) Rourkela in May 2008 and her master's degree in computer engineering from Texas A&M University, College Station in December 2011. Before joining the Ph.D. program at the University of Texas at Austin, she worked as a hardware design engineer in the SPARC processor group in Oracle, Austin. She has also completed several internships at AMD Research, Oracle labs and Texas Instruments.

Permanent address: Bhubaneswar, India

This dissertation was typeset with LaTeX[†] by the author.

[†] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.