Chip Multi Processor (CMP) is the most prevalent processor design today. The number of cores consolidated per chip has been rapidly growing as transistor density has been increasing in accordance to Moore’s law. With growing number of cores, multiple diverse workloads compete for the caches, memory controller (MC) and DRAM resources, collectively referred to as the memory hierarchy. This puts an increasing burden on the limited memory hierarchy resources making it the key bottleneck in CMPs. In turn, this shifts the CMP design focus from the core to the memory hierarchy.

CMPs present the design and research community with interesting challenges. With diverse design choices and multiple workloads competing for the memory hierarchy, design issues are growing exponentially in complexity. However, a CMP designer is still expected to produce a design that meets cost, power, schedule and performance constraints for target end user workloads. So, designers need an in-depth understanding of end user workloads, which either requires access to source code or some form of a detailed trace. However, end users are apprehensive about sharing code or traces due to the proprietary or confidential nature of code and data. Thus, on one hand designers need in-depth insights into such proprietary end user workloads, and on the other hand, end users cannot reveal any significant information about their code to designers. This is referred to as the proprietary workload problem. One example is a company with a proprietary trading algorithm that may be unwilling to share their code with designers. Another example is a highly classified weapons simulation code in a national lab which cannot be shared with most employees in the lab, let alone designers. A second problem
is the exponential growth in simulation time needed for exploring a wide design space for different workloads.

To bridge this gap, designers use a reduced representation of the code referred to as a clone. *Black box workload cloning* is an emerging approach that can solve the proprietary workload problem by creating a proxy for the proprietary workload. Cloning involves two key aspects: (1) Identifying key profiling statistics such that the statistics can faithfully represent the original workload. (2) Using the profiled statistics to generate a clone offline that can be in lieu of the original workload in the design environment. In this dissertation, we will investigate two black box cloning techniques that can be used to explore memory hierarchy design space. Although solving the simulation time problem is not the primary focus of cloning, a reduced representation of the original workload invariably reduces simulation time.

The first goal of this work is to solve the proprietary workload problem for the cache hierarchy in CMPs. We propose Workload Emulation using Stochastic Traces (WEST), a highly accurate black box cloning technique for replicating data cache behavior of arbitrary programs. First, we analyze what profiling statistics are necessary and sufficient to capture a workload. We show the importance of capturing temporal behavior over spatial or strided behavior. We found that traditional statistics like Stack Distance used for representing temporal accesses are necessary but insufficient. Second, we generate a clone stochastically that produces statistics identical to the proprietary workload. WEST clones can be used for exploring cache sizes, associativities, write policies, replacement policies, cache hierarchies and co-scheduling, at a significantly reduced simulation time. We use a simple IPC model to control the rate of accesses to the cache hierarchy. Our evaluation over a wide cache design space for single core and dual core CMPs shows that WEST can faithfully capture the behavior of the original workload. In addition
to solving the proprietary workload problem, WEST also reduces simulation time by an order of magnitude.

The second goal of this work is to create a cloning framework for studying and exploring MC/DRAM design space. We propose Memory Emulation using Stochastic Traces (MEMST), a black box cloning framework for accurately modeling MC/DRAM behavior. We provide a detailed analysis of statistics that are necessary to model a workload accurately from a MC/DRAM perspective. We found that temporal locality in MC/DRAM is a function of DRAM organization (rows, columns, ranks, banks, etc.) and various DRAM timing parameters (e.g. time to precharge, activate, read row, read column, etc.). Furthermore, MC/DRAM have their own design issues such as DRAM frequency, page policy, scheduling policy, etc that require unique statistics. We will also show how a clone can be generated from these statistics using a novel stochastic method. Finally, we will validate our framework across a wide design space by varying DRAM organization, address mapping, DRAM frequency, page policy, scheduling policy, input bus bandwidth, chipset latency, DRAM die revision, DRAM generation and DRAM refresh policy. Our evaluation of MEMST across a wide design space for single-core, dual-core, quad-core and octa-core CMPs showed that it can accurately mimic row buffer miss ratio, transaction latency, memory bandwidth and DRAM power of multi-program and parallel workloads. Similar to WEST, MEMST clones can also significantly reduce simulation time.
Workload Cloning: Emulating Memory Hierarchy Behavior using Stochastic Traces

by
Ganesh Balakrishnan

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2014

APPROVED BY:

Dr. Eric Rotenberg

Dr. James M. Tuck

Dr. Greg T. Byrd

Dr. Ken V. Vu

Dr. Yan Solihin
Chair of Advisory Committee
DEDICATION

To my wife, Saradha and my children, Shraddha and Sneha for their support and patience over the years. Second, to my grandmother Jayalakshmi Rajaram who was a source of inspiration throughout my life.
BIOGRAPHY

Ganesh Balakrishnan was born in the steel city of Bhilai, India in 1977. He earned his Bachelor’s degree in Applied Electronics and Instrumentation Engineering from National Institute of Technology, Rourkela in 1999. He pursued his Master’s degree in Electrical and Computer Engineering from Purdue University, West Lafayette, Indiana. After graduating in 2001, he joined IBM’s System and Technology Group as an engineer in the Network Processor performance team. He moved to the System x and BladeCenter performance group in 2003 and worked on processor and networking performance on multiple generations of servers. He moved to the System Architecture and Performance team in 2013 and has been working on various aspects of performance in future POWER processors. He started his PhD in 2009 while still at IBM in the Center for Efficient, Scalable and Reliable Computing (CESR) at North Carolina State University.
ACKNOWLEDGEMENTS

I would like to first express my appreciation for my research advisor, Dr. Yan Solihin. He was instrumental in shaping my research through his astute guidance, insightful thoughts and constant support. I also enjoyed our intellectually challenging brainstorming sessions. I also cherished his role as a good friend which helped make my research more enjoyable.

I am also thankful to my PhD advisory committee members, Dr. Eric Rotenberg, Dr. James Tuck and Dr. Greg Byrd for providing me valuable feedback which helped me define the scope of my research. I would like to specifically thank Dr. Rotenberg. I took a few courses with him as a student which left an indelible impression on me. Besides being an outstanding speaker and motivator, his dedication and enthusiasm as a teacher and attention to detail have inspired me. He is a great role model for students, aspiring professors and computer architects.

I would like to also thank my fellow ARPERS and CESR students. Special thanks to my colleagues, George Patsilaras and Ahmad Samih, for their countless hours of brainstorming.

My PhD would not have been possible without the support of my management and colleagues at IBM. A special thanks to my managers, Kanton Reynolds and Ken Vu for their continued support over the years.

Getting a PhD is largely a struggle and is not possible without strong support from the family. I would like to thank my parents, K. Balakrishnan and Girija Balakrishnan, for their constant encouragement throughout my life and for instilling the value of good education. I started my PhD part-time in 2009 without fully appreciating the enormity of the task ahead of me. A PhD along with a full work life places enormous strain on the family. My daughters, Shraddha and Sneha, have seen me working on my research since
they were babies and hopefully, we can make up for the lost time in the years to come.
And lastly, I reserve my most special appreciation for my wife, Saradha, without whom
I could not have started my pursuit of a PhD, let alone work towards completion.
### TABLE OF CONTENTS

**LIST OF TABLES** .......................................................... viii

**LIST OF FIGURES** ......................................................... ix

**Chapter 1 Introduction** ................................................ 1

1.1 WEST: Cloning Data Cache Behavior using Stochastic Traces ...... 6
1.2 MEMST: Cloning Memory Behavior using Stochastic Traces ......... 10
1.3 Organization of the Dissertation ...................................... 13

**Chapter 2 WEST: Workload Emulation using Stochastic Traces** .... 14

2.1 WEST Framework ....................................................... 14
2.1.1 Scope and Limitations ............................................ 16
2.1.2 Analysis of Necessary and Sufficient Statistics ................. 16
2.1.3 Generation Phase ................................................ 25
2.1.4 Trace Clone Generation ......................................... 27
2.1.5 Binary Clone Generation ....................................... 32
2.2 Evaluation Methodology .............................................. 32
2.3 Results and Analysis ................................................ 35
2.3.1 Results on Single Core (System A) ............................ 37
2.3.2 Results for Co-schedules on CMPs (System A) ................. 40
2.3.3 Results on Single Core (System B) ............................ 43
2.3.4 Impact of Trace Length on Convergence ....................... 44
2.4 Related Work .......................................................... 46
2.5 Conclusions ............................................................ 48

**Chapter 3 MEMST: Memory Emulation using Stochastic Traces** .... 49

3.1 Background ............................................................ 49
3.1.1 DRAM Organization ............................................. 49
3.1.2 Page Policy ....................................................... 53
3.1.3 Scheduling Policy ............................................... 54
3.1.4 Address Mapping Policy ....................................... 54
3.2 MEMST Framework ................................................... 56
3.2.1 Analysis of Necessary Statistics ............................... 57
3.2.2 Clone Generation ............................................... 69
3.3 Evaluation Methodology ............................................. 77
3.4 Results and Analysis ................................................. 80
3.4.1 Design Space Exploration: Multiprogram workloads ........... 80
3.4.2 Design Space Exploration: Parallel workloads .......................... 88
3.4.3 Sensitivity Study: Simulation Time ........................................ 89
3.5 Related Work ........................................................................... 89
3.6 Conclusions ............................................................................. 91

Chapter 4 Conclusion ...................................................................... 92

References ..................................................................................... 94
LIST OF TABLES

Table 2.1 System Configuration for System A ....................................... 33
Table 2.2 System Configuration for System B ................................. 34
Table 2.3 Selection of benchmarks for evaluation of co-scheduling (Benchmarks in bold selected) .................................................. 41

Table 3.1 System A: Multi-program Workloads .................................. 77
Table 3.2 System B: Parallel Workloads ............................................ 78
Table 3.3 MC and DRAM Design Space ............................................. 78
Table 3.4 Benchmark Mixes ............................................................ 79
Table 3.5 Effect of reducing the number of references ....................... 89
# LIST OF FIGURES

| Figure 1.1 | Temporal and Spatial reuse at 64B block granularity shown with various strides. The machine configuration follows Table 2.1 | 8 |
| Figure 2.1 | WEST Framework | 15 |
| Figure 2.2 | Global vs. Set Stack Distance in a 4-way cache | 20 |
| Figure 2.3 | Set Reuse profile with a depth of 8 in a 32KB L1 cache | 21 |
| Figure 2.4 | Fraction of writes across sets and stack positions | 22 |
| Figure 2.5 | Set Access Distribution (SAD) in a 32KB L1 cache with 128 sets | 22 |
| Figure 2.6 | Co-scheduled behavior of 2 copies of hmmer in a 2M shared cache | 25 |
| Figure 2.7 | Trace Generation Algorithm for a single memory reference (a), contents of various arrays before address generation (b) versus after address generation (c) | 31 |
| Figure 2.8 | Comparison of miss ratio errors of GSD vs. WEST clones (Section 2.1.2) for various L1 cache configurations | 36 |
| Figure 2.9 | System A L1/L2 Cache Size DSE | 37 |
| Figure 2.10 | System A Set Change DSE | 38 |
| Figure 2.11 | System A Additional Cache Level | 39 |
| Figure 2.12 | System A Write Policy DSE | 40 |
| Figure 2.13 | System A Replacement Policy DSE | 40 |
| Figure 2.14 | Impact of varying L2 cache size on L2 miss ratio: benchmark 0 (left) & benchmark 1 (right) | 42 |
| Figure 2.15 | Impact of varying L2 cache size on IPC: benchmark 0 (left) & benchmark 1 (right) | 43 |
| Figure 2.16 | Impact of varying L1 cache size on L2 miss ratio: benchmark 0 (left) & benchmark 1 (right) | 44 |
| Figure 2.17 | Impact of varying L1 cache size on IPC: benchmark 0 (left) & benchmark 1 (right) | 45 |
| Figure 2.18 | System B (Out-of-order) Evaluation | 45 |
| Figure 2.19 | Trace Length Analysis | 46 |
| Figure 3.1 | DRAM Physical Hierarchy. Additional chips for reliability not shown for simplicity | 51 |
| Figure 3.2 | DRAM Logical View | 52 |
| Figure 3.3 | Address Mapping Schemes | 55 |
| Figure 3.4 | MEMST Framework | 57 |
| Figure 3.5 | DRAM addressing and analogy to cache addressing | 59 |
| Figure 3.6 | Logical view of DRAM | 61 |
| Figure 3.7 | MEMST profiling structure consists of a stack of Row ID Tables (RIT) | 61 |
Figure 3.8  Example of inter-arrival gap showing an interleaved stream from two cores to a single row in a RIT. Inter-arrival gap shown between successive transactions. Updates to timing distributions for core0 and core1 shown below. ................................................................. 64
Figure 3.9  Example of HPA for two streams of memory references. ............. 64
Figure 3.10  Example of HIP for a stream of memory references from a single core to a single row in the RIT. References interrupted by accesses from another core shown with arrows. ...................................................... 66
Figure 3.11  Symbiotic hits for parallel workloads ........................................ 67
Figure 3.12  Example of an interleaved stream of references from three cores. The table shows $TI[c_x][c_y]$. ................................................................. 68
Figure 3.13  Example of two standalone streams from two cores interacting in a co-schedule. ................................................................. 70
Figure 3.14  State Diagram driving address generation. States generating addresses are highlighted. ................................................................. 76
Figure 3.15  Multi-program workload evaluation: DRAM Organization DSE .... 81
Figure 3.16  Multi-program workload evaluation: DRAM Address Mapping DSE 81
Figure 3.17  Multi-program workload evaluation: DRAM Frequency DSE ....... 82
Figure 3.18  Multi-program workload evaluation: MC Transaction Scheduling Policy DSE ................................................................. 83
Figure 3.19  Multi-program workload evaluation: MC Page Policy DSE ......... 84
Figure 3.20  Multi-program workload evaluation: DRAM Generation DSE .... 85
Figure 3.21  Multi-program workload evaluation: Bus Bandwidth and Chipset Latency DSE ................................................................. 86
Figure 3.22  Multi-program workload evaluation: DRAM Refresh Policy DSE ... 87
Figure 3.23  Parallel workload evaluation .................................................. 88
Chapter 1

Introduction

Transistor density in processors is growing in accordance with Moore’s law [37] and this trend is expected to continue in the processor industry. Historically, processor performance has largely focused on single core performance and the increasing transistor density, coupled with innovations in microarchitecture, compiler and circuit technology fueled this trend. However, this trend was stalled by the power wall and the growth in transistor density instead ushered in the era of multicore computing in the early 2000s with multiple cores consolidated per die. Today, Chip Multi Processors (CMP) dominate the landscape of computing. The rapid growth in the number of cores has accelerated workload consolidation with diverse workloads competing for cache resources, memory controller (MC) and DRAM-based memory. These highly contended resources are becoming the major bottlenecks in CMP performance. We collectively refer to the cache, memory controller and DRAM-based memory as the memory hierarchy. In addition to being a major performance bottleneck, the memory hierarchy also offers a wide range of design choices. For example, cache designers are presented with different options in cache organization, replacement policy, write policy, etc. MC/DRAM designers have a design
space that includes choices in DRAM organization, address mapping policy, scheduling policy, etc. Hence, the design focus has been rapidly shifting from the core to the memory hierarchy.

To effectively evaluate the vast memory hierarchy design space, designers need to carefully select workloads. The choice of workloads is instrumental in driving key design decisions that influence performance, cost, power and schedule. Ideally, evaluation techniques should use workloads that are representative of end user workloads. Using an end user workload bolsters the confidence of both the designer and the end user because it helps the designer make workload-centric trade-offs and harden the design. However, end users are generally unwilling to share their workloads with designers. Access to source code is highly restricted due to the proprietary nature of applications. One example is a company with a proprietary trading algorithm that may be unwilling to share their code with designers. Another example is a highly classified weapons simulation code in a national lab which cannot be shared with most employees in the lab, let alone designers. A possible alternative to source code access is collecting a trace. Again, this is not a viable option because traces are generally very large in size and it is not possible for the end user to determine whether there is any confidential data captured in the trace. So, most end users are apprehensive about sharing code or traces due to the proprietary or confidential nature of code and data [21, 30]. This is called the proprietary workload problem. Thus, on one hand designers need in-depth insights into end user workloads, and on the other hand, end users cannot reveal any significant information about their code to designers.

So, in lieu of direct access to source code or traces, designers have a few alternatives in their evaluation toolkit. One possibility is to use industry-standard benchmarks for simulations. Industry-standard benchmarks like Whetstone [15], SPEC CPU2006,
SPECjbb2005 [7], TPCC [6], etc. have evolved over time in an effort to provide a common framework to measure and compare performance. It is very common for designers to use such benchmarks during the early design phase in a simulation environment. However, benchmarks may not be either representative of proprietary workloads or may not exist at all [32, 30]. End users attach far more credibility to data collected using their workload as opposed to standard benchmarks [30]. So, industry-standard benchmarks do not address the issue of proprietary workloads. Another alternative to detailed simulations is statistical simulation [17, 18, 16, 26, 27, 38, 39]. Statistical simulation generates a short synthetic trace from a statistical profile of workload attributes such as branch misprediction rate, cache miss ratio, etc. The synthetic trace can be simulated using a statistical simulator to provide a performance estimate. The main benefit of this approach is reduced simulation time. However, there are two important drawbacks of this technique. Firstly, the synthetic traces can only be used in statistical simulators not execution-driven simulators or real hardware. Secondly, their purpose is to cull very early designs and not to solve the proprietary workload problem. Another alternative is to build analytical models [33, 28, 19] to estimate performance. While analytical models can produce results quickly, it requires considerable effort to represent complex microarchitectures with mathematical models. When the underlying architecture is changed, the analytical model has to be redesigned. So, similar to statistical simulation, they are generally used in very early design. So, the existing techniques do not solve the proprietary workload problem and can not be used for detailed design work.

One promising approach to this problem is workload cloning. Workload cloning attempts to create a clone (binary or trace) of the proprietary workload that produces the same microarchitectural attributes as the proprietary workload. The clone binary or trace can be run on an execution-driven simulator or a trace-driven simulator.
There are generally two approaches to cloning a workload: *white box* or *black box*. In a white box approach, subject experts gather as much information as they can from the end user, such as code behavior, algorithm, data characteristics, etc. and use the information to create a clone that hopefully produces behavior similar to the end user workload. The advantage of the white box approach is that it can ultimately yield an accurate clone, due to the accumulated experience of the subject experts, and feedback from the end users. The drawbacks of the white box approach are that it is highly manual, requires years of accumulated expertise and experience, and is riddled with potential pitfalls. For example, even with the knowledge of a particular graph algorithm, subject experts may still be unable to reproduce the cache behavior of the workload unless they use the correct data structure and layout in memory.

In contrast, the black box approach works without requiring subject knowledge about the workloads. Workloads are profiled to obtain key statistics, and a clone that can produce similar statistics is generated. The key advantage of the black box approach is its simplicity: it can be automated and does not require subject experts. However, there are substantial challenges to the black box approach. The first challenge is to identify key statistics that *necessarily* and *sufficiently* represent the workload’s behavior. For example, the number of instructions is neither necessary nor sufficient in representing a workload’s data cache behavior. The cache miss ratio is necessary but not sufficient in representing a workload’s cache behavior. A cache miss ratio for a particular cache configuration does not provide any information about what the miss ratio would be for other cache configurations. So, if a clone generates an address stream such that it produces the same miss ratio for a given cache size, there is no guarantee that the same clone will produce the same miss ratio as the original workload on different cache sizes. Profiling is done by either instrumenting the application binary or by gathering a trace and then profiling
it or by using hardware analyzers/profilers. Whatever the profiling method may be, the only information exchanged between the end user and the design team is the workload profile. This solves the proprietary workload problem. The second challenging aspect of cloning is generation of the clone in the design environment using the profile provided by the end user. The clone must not only be able to accurately mimic the behavior of the profiled workload as the underlying architecture is changed, but should also be micro-architecture independent and portable. Micro-architecture independence is achieved by selecting profiling statistics that are agnostic to the underlying hardware. For example, if statistics are selected that are dependent on the cache replacement policy, then the proprietary application has to be profiled again when the replacement policy is changed. This implies that statistics should model application behavior and not micro-architectural behavior. Portability of clones is achieved by using high-level languages like C, C++ or assembly-level directives.

The focus of this dissertation is black box workload cloning for exploring the memory hierarchy design space. We will first analyze statistics that are necessary to sufficiently capture application behavior. These statistics are collected when profiling workloads that may be proprietary in nature. We will also look at clone generation techniques, where we will show how a workload clone (trace/binary) can be generated using the collected profile. Finally, we will validate our profiling and clone generation techniques through extensive design space exploration of the underlying architecture.

Cache design is one of the critical aspects of processor architecture. Caches play a significant role in the overall performance of the system because they bridge the performance gap between the processor core and the MC/DRAM subsystem. Cache designers are presented with numerous design choices like cache organization (size and associativity), replacement policy, write policy, cache hierarchy (shared/private), etc. The first
goal of the dissertation is to build a cloning framework that enables cache designers to study and explore the cache design space using clones that are representative of end user workloads.

MC/DRAM is another important component in processor design. MC and DRAM-based memory are generally shared among multiple cores in a CMP. The bandwidth to main memory does not grow at the same pace as the number of cores due to pin limitations, power constraints or packaging costs [41]. So, as the number of cores increase, the contention for memory bandwidth also increases making it one of the key bottlenecks in the system. So, designers need a framework that allows them to effectively evaluate the numerous MC/DRAM design choices like memory organization, scheduling policy, page policy, DRAM frequency, etc. The second goal of the dissertation is to provide a cloning framework for modeling MC/DRAM behavior for proprietary workloads.

1.1 WEST: Cloning Data Cache Behavior using Stochastic Traces

With the advent of Chip Multi Processors (CMP) and multiple workloads competing for the on-die cache, cache design is becoming very challenging. Fundamentally, an ideal cache should be optimized for end user workloads while meeting cost, power and schedule constraints. So, cache designers need an in-depth understanding of end user workloads, which either requires access to source code or some form of a detailed trace. But, due to the proprietary nature of code or data, cache designers can rarely get access to such workloads. One approach to solving the problem is the black box workload cloning technique. Using such a technique, the proprietary workload is first reduced to a set
of statistics and then using these statistics, a clone is generated using an automated method. The clone can be used by the designer in place of the proprietary workload for studying cache designs.

Bell et al. [10] and Joshi et al. [30] are credited with pioneering work in the area of black box cloning for instruction level parallelism behavior. They create clones by generating instructions that replicate the original workload’s statistics in terms of instruction mixes, data flow graphs, and data dependence distance. Their contribution inspires this work into looking at whether we can clone data cache behavior accurately. While their work accurately captures instruction behavior, the data cache accesses are abstracted as a single dominant stride [30]. We found, however, that a single dominant stride is insufficient for representing a workload’s data cache behavior. Figure 1.1 shows the temporal and spatial reuse of CPU2006 [7] and BioBench benchmarks [8]. To characterize the memory references, we detected stride patterns at a cache block granularity for every memory reference and classified them based on the stride amount (stride amount of zero is classified as temporal reuse). The figure shows that for about 2/3 of the benchmarks, at least 90% of the accesses are temporal in nature. It also shows that for the remaining benchmarks, there are at least 3 dominant strides (>1% of total access) that account for the memory accesses. One of the benchmarks, tiger, has over 80% of memory accesses that are random in nature. Hence, capturing one dominant stride is insufficient to fully capture arbitrary access patterns for many workloads.

In this work, we propose **Workload Emulation using Stochastic Traces (WEST)**, a black box workload cloning technique for replicating data cache behavior. A key contribution of this work is an analysis of what statistics are necessary and sufficient to capture the behavior of arbitrary memory access patterns. In particular, a stack distance profile
Figure 1.1: Temporal and Spatial reuse at 64B block granularity shown with various strides. The machine configuration follows Table 2.1.

... was long believed to sufficiently capture any memory access pattern, and was used by various analytical models for predicting and explaining various cache performance phenomena, e.g. studies in [13, 14, 28, 34]. We show, however, that stack distance profile is necessary, but not sufficient. We found that in addition to stack distance profile, temporal locality across sets, access distribution across sets, and read/write distribution, need to be profiled at a fine granularity in order to capture data cache behavior. The combination of all these profiled statistics capture arbitrary memory access patterns very well. This is demonstrated in the following way: statistics profiled from one cache configuration enables the generation of a clone that produces nearly identical data cache statistics on multitudes of other single core cache configurations, such as different cache sizes, cache associativities, cache sets, cache write policies, replacement policies; and also a shared cache configuration in a CMP architecture.

Secondly, rather than approximating the profiled statistics, WEST produces trace or binary clones that are guaranteed to produce statistics identical to the original workloads by using a strict convergence approach. A workload and a clone are said to be statistically
identical if they produce the same statistics under the machine configuration from which the statistics are profiled.

We evaluated WEST using the CPU2006 and BioBench benchmark suites over a wide cache design space for single core and dual core CMPs. In comparison to the profiled workloads, the clones, on average, had an error of 0.4% for miss ratio for 1394 single core cache configurations and 3.1% for miss ratio for over 600 co-scheduled configurations.

Other benefits of WEST include clone portability and short simulation time. Clones are highly portable because they are written using standard C-code and assembly language directives. WEST clones speed up execution by shrinking the number of instructions in the original workload which range from millions to billions by orders of magnitude and still produce identical statistics.

To summarize, the key contributions of this work are:

1. An analysis of what key statistics are necessary and sufficient to represent data cache behavior.

2. A framework to generate statistically-identical clones to reproduce data cache behavior.

3. An evaluation that demonstrates that clones generated using one cache configuration, can produce closely matching miss ratios and IPCs of the original benchmarks, on a wide range of other cache configurations, including a variety of sizes, associativities, number of sets, write policies, replacement policies and different levels of caches.

4. An evaluation that demonstrates that clones can be co-scheduled on CMPs to produce closely matching cache miss ratios and IPCs compared to co-schedules of
the original benchmarks.

1.2 MEMST: Cloning Memory Behavior using Stochastic Traces

The memory subsystem has been a key bottleneck in Chip Multi Processors (CMP) in terms of performance and power. With multiple workloads competing for memory capacity and bandwidth, Memory Controller (MC) and DRAM design is becoming very challenging. A good design should optimize for various end user workloads while meeting cost, power and schedule constraints. The importance of using end user workloads for characterizing and exploring design space instead of using industry-standard benchmarks is well-known [30, 20].

MC/DRAM designers need an in-depth understanding of end user workloads. Historically, this has required either access to the source code or a trace. Getting access to source code or traces from the end user is rarely possible due to the proprietary nature of the source code or the confidential nature of data [9, 21, 30].

To solve this problem, designers use a reduced representation of the proprietary code. This miniaturized version is referred to as a clone. There are two approaches to cloning a workload: white box or black box [9]. The black box technique provides a convenient way to abstract the workload to a set of key statistics instead of needing detailed understanding of underlying code structure or algorithm. So, such cloning techniques are generally preferred for solving the proprietary workload problem.

Prior black box cloning techniques [10, 30, 21, 9, 24] target different aspects of system design. The cloning framework developed by Bell and John [10] and Joshi et al. [30]
models Instruction Level Parallelism (ILP) behavior. Balakrishnan and Solihin proposed a WEST framework to clone data cache behavior. Ganesan et al. [21, 24] extended the framework proposed by Bell et al. to capture Memory Level Parallelism (MLP) behavior and clone multi-threaded workloads. Prior cloning techniques have a limitation that they were not designed and cannot be used to extensively study and explore MC/DRAM design space. Cloning techniques in [10, 21, 24, 30] model the spatial locality of memory accesses by abstracting them as a single dominant stride per static load/store, but do not model temporal locality of memory behavior [9]. While WEST captures temporal locality in caches, WEST abstracts cache miss streams as random references, hence it does not capture temporal and spatial locality beyond the last level cache.

In this work, we propose Memory EMulation using Stochastic Traces (MEMST), a black box workload cloning technique designed for replicating MC/DRAM behavior. Finding architecture-independent statistics that model MC/DRAM performance and power is challenging. MC/DRAM performance is dependent on DRAM timing parameters (e.g. tRP, tRCD, tCAS, etc.), scheduling policy and page policy. Secondly, locality in MC/DRAM is a function of DRAM organization (rows, columns, ranks, banks, etc.) and address mapping policy. Thirdly, DRAM has timing attributes like (tFAW and tRRD) that affect the power consumed by DRAM. Hence, MC/DRAM performance and power are dependent on aspects that are different from caches. Consequently, we found that the statistics that we need to capture to clone memory access behavior at the MC/DRAM level are more numerous and complex than those at the cache level. For example, our statistics include not just the number of hits or misses/conflicts, but also probability distributions of various timing characteristics, such as the time between row buffer hits and misses, burst of hits, burst lengths, and interleaving of bursts from different threads in a multi-programmed or parallel workload. Using these statistics, MEMST can generate
clones that can accurately mimic the original workload across a wide design space that includes varying organization, DRAM page sizes, ranks, channels, DDRx frequencies, MC page policy, MC scheduling policy, DRAM die revisions, DDRx generations and refresh policies. The analysis of statistics needed to replicate MC/DRAM behavior is the first contribution of this work.

A second contribution of this work is a unique clone generation technique. Modeling co-scheduling behavior in MC/DRAM requires generating parallelism at the thread-level and bank-level and ensuring appropriate serialization of requests to the same bank. This requires ensuring fine-grained coordination between the clones running on different cores to model co-scheduled behavior accurately. To achieve this, we propose to generate memory references using a parallel program and synchronize the threads using baton locks so that the timing of the arrivals and how accesses from different threads are interleaved are faithfully replicated.

One by-product of cloning is that MEMST can shrink the number of accesses of the original workload to a small fraction and still maintain a high degree of accuracy. This leads to much shorter simulation times, making design space exploration cheaper for MC/DRAM designers.

We evaluated MEMST using the CPU2006, BioBench, Stream and PARSEC benchmark suites over a wide design space for single-core, dual-core, quad-core and octa-core CMPs. The clones were able to accurately represent the benchmarks for over 7900 data points with an average error of 1.8% and 1.6% for transaction latency and DRAM power respectively.

To summarize, the key contributions of this work are:

1. An analysis of statistics needed for accurate clone generation that reproduces row
buffer miss ratio, memory bandwidth, transaction latency and DRAM power for proprietary workloads.

2. A novel cloning methodology to generate clones for exploring a wide MC and DRAM design space in simulators and hardware.

3. A comprehensive evaluation to validate the cloning technique in single-core, dual-core, quad-core and octa-core CMPs. The design space includes varying DRAM organization, address mapping, DDRx frequencies, MC page policy, MC scheduling policy, DRAM die revisions, DDRx

1.3 Organization of the Dissertation

The rest of the report is organized as follows. Chapter 2 explores workload cloning for studying data cache hierarchy in CMPs. In this work, a black-box profiling and clone generation framework is presented. Chapter 3 provides a detailed cloning framework to study and explore a wide design space in memory controllers and DRAM-based memory. Finally, Chapter 4 concludes this work.
Chapter 2

WEST: Workload Emulation using Stochastic Traces

This chapter is organized as follows: Section 2.1 describes the profiling technique and clone generation methodology. Section 2.2 describes our evaluation methodology. Section 2.3 provides the results from our evaluation and analyzes the findings. Section 2.4 describes related work. Section 2.5 concludes this work.

2.1 WEST Framework

The goal of our Workload Emulation using Stochastic Traces (WEST) framework is to model data cache behavior for individual and multi-programmed sequential workloads. The input to WEST is either a proprietary workload, a benchmark, or a user-defined program. WEST can output a stochastic trace, i.e. synthetic memory references that mimic the original data cache behavior, or a binary, i.e. a stand-alone executable that generates the memory references when it runs. We will refer to the output as a trace
clone, a binary clone, or simply as a clone when it applies to both types.

Components of the WEST framework are shown in Figure 2.1. The profile must contain necessary and sufficient statistics needed for accurate clone generation. Custom user-defined profiles are also possible for generating clones for validation and debugging. The profile is fed to the trace generator (c) and the binary generator (d). The trace generator outputs a stream of memory references (e) that can be used in trace-driven simulators (g), while the binary generator creates an intermediate C program using the profile. The C program is then compiled to generate a binary (f), that can be used in full system simulators or real hardware (h).

Figure 2.1: WEST Framework.
2.1.1 Scope and Limitations

WEST aims to model data cache behavior for sequential programs running alone or co-scheduled on a CMP. Currently, WEST does not capture TLB behavior, coherence misses in parallel programs, instruction cache behavior, and cache pollution from instructions. Our primary goal is to match the miss ratio and IPC of the profiled workload when the underlying cache configuration is changed. We assume a simple processor model for IPC, using IPC as a mechanism to control rate of issue of accesses to the memory hierarchy. Producing IPC for a more complex, OoO processor model, requires not only reproducing data cache behavior, but also instruction level behavior, which is beyond the scope of this work. The data cache behavior cloned by WEST depends on the time period for which statistics are profiled. WEST can clone whole-program behavior as well as phase-specific behavior, if profiles are collected for each phase. WEST does not collect statistics for data cache behavior within a cache block, hence different profiling runs and clones are needed for different cache block sizes. Finally, we ignore the contribution of compulsory cache misses.

2.1.2 Analysis of Necessary and Sufficient Statistics

As discussed earlier, the black box approach to workload cloning does not require subject expert knowledge and can be automated. However, one of the key challenges is to ensure that the profiled statistics that are used for generating the clone necessarily and sufficiently represent the workload’s behavior. This section discusses our analysis of what statistics fully capture a workload’s data cache behavior.

As a starting point, a sequential application’s data cache behavior consists of spatial reuse and temporal reuse. Intuitively, capturing and mimicking them allows the original
data cache behavior of the application to be cloned. However, this fact still does not provide us an insight into how spatial reuse and temporal reuse can be measured, and what statistics are sufficient to represent them. Let us examine the two questions in more detail.

Spatial reuse represents the likelihood of accessing a consecutive datum in the near future. Prior studies in benchmark cloning attempt to discover a stride amount $s$ such that when address $x$ is accessed, $x + s$ is the likeliest address to be accessed next by the program [30]. While such an approach has merit, Figure 1.1 shows that it is rare for arbitrary programs to exhibit memory access patterns with a dominant stride amount, especially at a cache block granularity. We will further argue that spatial reuse is subsumable by temporal reuse, consequently making it unnecessary to capture spatial reuse. There are several reasons for this.

First, intuitively, within a cache block, spatial accesses to adjacent words in the block will appear as multiple accesses to the same block. Thus, spatial reuse at a fine granularity manifests as temporal reuse at a coarser granularity. Secondly, any spatial block that is accessed, is also a block that has been accessed before, unless when the block is first accessed. This means that aside from the initial accesses, any access to a word or block is a temporal reuse of the word or block. Hence, excluding the initial accesses, spatial reuse is subsumed by temporal reuse. However, the converse is not true. Some (or much) temporal reuse is not spatial reuse. Hence, capturing spatial reuse only such as the stride amount is clearly incomplete. Excluding the initial accesses is fine in most situations as they only cause compulsory cache misses, which are overwhelmed by orders of magnitude by other (capacity and conflict) cache misses.

The next question is what statistics can sufficiently capture temporal reuse. Traditionally, temporal reuse has been measured in terms of stack distance distribution. Stack
distance measures the distribution of the number of distinct data elements accessed between two consecutive references to the same element. Mattson et al. originally proposed stack distance profile as a way to measure misses for various storage structure sizes in one pass [36], for a fully-associative structure employing LRU replacement policy. Hill showed that stack distance profiling is also applicable to a set-associative cache, if the profile is collected for each set in the cache [29]. Many subsequent studies use stack distance profiles to predict various cache performance phenomena. For example, stack distance profile was used to predict the cache miss impact of cache sharing in CMP [13], predict the impact of prefetching on context switch misses [34], predict the performance of various cache replacement policies [28], predict throughput [14], and many others. The prevalence of the use of stack distance profile as the key statistic for predicting cache performance behavior indicates that conventional wisdom assumes that temporal reuse can be fully captured by the stack distance profile. In this work, we argue that, contrary to conventional wisdom, additional statistics are needed for fully capturing temporal reuse pattern of an application.

Let us start by reviewing what a stack distance profile captures. In a stack distance profile, the LRU stack position for each block is kept current at all time. For a cache with $S$ sets, let $C_{ij}$ denote the number of times data block located in the $i^{th}$ set and $j^{th}$ stack position is accessed, where $i = 1, 2, \ldots, S$ and $j = 1, 2, \ldots, \infty$. For simplicity of discussion, we assume that the stack distance profile keeps track of an unlimited number of ways.

To capture the statistical behavior of an application, it is often better to represent the stack distance profile as a probability distribution. In conventional uses [13, 14, 28, 34], stack distance profile keeps one set of values $GSD(j)$ where $j = 1, 2, \ldots, \infty$, corresponding to different stack positions, shared by all the cache sets. We will refer to it as Global
**Stack Distance** (GSD) profile. We argue that despite its prevalence, conventional GSD profile provides insufficient information about the temporal reuse behavior. We use a subtly different stack distance profile, **Set Stack Distance** (SSD), that keeps one set of private values for each cache set, in order to capture temporal reuse behavior for each individual set. More formally, $SSD(i, j)$ and $GSD(j)$ can be expressed as:

$$SSD(i, j) = \frac{C_{ij}}{\sum_{j=1}^{\infty} C_{ij}}$$

$$GSD(j) = \frac{\sum_{i=1}^{S} C_{ij}}{\sum_{i=1}^{S} \sum_{j=1}^{\infty} C_{ij}}$$

The reason to use SSD in lieu of GSD is due to the large variation of temporal reuse across sets. We illustrate this variation in Figure 2.2, which shows that the deviation in fractions of accesses that fall into a particular stack position can be very substantial: all 4 benchmarks show substantial deviation in the MRU position (> 18%) and in the miss position (> 8%). In the intermediate positions, cactus shows large deviations in all positions, astar shows only in the MRU-1 position while bwaves shows no major deviation. While only four benchmarks are shown, large deviations are common in other benchmarks. Therefore, GSD is insufficient for capturing the data cache behavior of an application, and SSD should be used instead.

However, even SSD is necessary but insufficient to capture an application’s temporal behavior. This is because of our observation that there is a very strong locality **across cache sets**, where recently visited sets are more likely to be revisited. Figure 2.3 shows distribution of accesses to eight most-recently visited sets. The figure shows that indeed, temporal reuse is present across cache sets with an average of 30% of accesses to the most recently visited set. We will refer to this as **Set Reuse** (SR) Profile.

To perform SR profiling, we keep the last $H$ sets that are visited in the cache, sorted based on their recency order, and increment the counter anytime a set that is accessed
matches any of the $H$ sets. Let $SR(i)$ denote the number of times (in percent) that an access is to the same set as in the $i^{th}$ position in the window of $H$ most recently visited sets, for $i = 1, 2, \ldots, H$. Let $SR(H + 1)$ denote the number of times (in percent) that an access is to a set that is different than the most recent $H$ sets. In profiling, we use $H = 8$, because the reuse across sets falls sharply beyond this value.

Another problem with stack distance profile is that it does not distinguish between read and write accesses. However, to clone the data cache behavior, we must distinguish between read and write accesses. For caches that form a multi-level cache hierarchy, write access in an upper level cache affects the access pattern in a lower level cache. For example, suppose that the L1 data cache is write through. A read access to a block in the L1 data cache does not affect the L2 cache. However, a write access to a block in the L1 data cache is written through to the same block in the L2 cache, causing the block to become the most recently used block in the L2 cache. The impact of writes on the recency order of the lower level cache also determines which block will be evicted, and ultimately the number of cache misses.
For these reasons, cloning data cache behavior requires reads and writes to be distinguished. The remaining question is what statistics should be collected. There are three possibilities: fraction of accesses that are writes for the entire cache (global), fraction of accesses that are writes for each stack position for the entire cache (per-distance), and fraction of accesses that are writes for each set and each stack position (per-set). A global write fraction is sufficient if the write fraction is independent of stack position and does not vary across sets. A per-distance write fraction is sufficient if there is no variation across sets. We choose per-set write fraction because as shown in Figure 2.4, there is a significant variation in fraction of writes across stack positions and across sets (shown by the scatters), which cannot be captured by global or per-distance write statistics. We refer to the per-set profile as Write Fraction (WF).

Besides SSD, SR, and WF probability distributions, we also need to collect the distribution of accesses to different sets, which we will refer to as Set Access Distribution (SAD). SAD can be easily computed from SSD profile counters. More specifically:

$$SAD(i) = \frac{\sum_{j=1}^{\infty} C_{ij}}{\sum_{i=1}^{S} \sum_{j=1}^{\infty} C_{ij}}.$$
As illustrated in Figure 2.5, certain sets are accessed far more frequently than others, with some sets receiving as much as 30% of the accesses. We observe similar behavior across all benchmarks, but only ten representative benchmarks are shown.

Figure 2.5: Set Access Distribution (SAD) in a 32KB L1 cache with 128 sets

We also capture SSD, SR, WF, and SAD profiles for all profiled cache levels because it determines whether a clone that is generated while assuming a certain cache hierarchy,
will still reproduce data cache behavior when we add or subtract levels of caches, or when we change the configuration of the upper level caches which filter out accesses to the lower level caches (more in Section 2.3). So, the clone is generated to capture temporal reuse at all profiled cache levels. We will show that although we profile using an L1 and L2 cache, we can accurately predict miss ratios even when we change cache sizes to vary the amount of filtering from the higher level caches, change the number of sets at lower cache levels and add additional cache levels. An important question is whether SSD, SR, WF, and SAD should be collected for the different caches separately. This question determines whether the profiling should be performed at the L1 cache only, L1 and L2 caches, L1/L2/L3 caches, etc. The reason why this question is important is not because of profiling complexity since all the profiling statistics can be collected from a single execution/simulation run. The reason why this question is important is because it determines whether a clone that is generated while assuming a certain cache hierarchy will still reproduce data cache behavior when we add or subtract levels of caches, or when we change the configuration of the upper level caches which filter out accesses to the lower level caches.

To answer the question, we note that in our profiling, we capture the SSD, SR, WF, and SAD of both the L1 cache and the L2 cache, since they are already available under the configuration used to collect the profile for clone generation. As a result, when we change the L1 or L2 cache sizes, the clone produces nearly-identical statistics with the original benchmarks (more in Section 2.3). However, we experimented with configurations that we know will change the SSDs, such as changing the number of cache sets (which makes an SSD split into multiple SSDs, or combines several SSDs into one), and adding a new cache level (for which its SSD was not considered for clone generation). In these cases, our experimental results show that WEST clones produce slightly higher errors, but the
errors remain very small.

So far, we have made the case that SSD, SR, WF, and SAD are required to fully capture the temporal reuse pattern of a workload. These statistics are sufficient for a sequential workload running on a single core system. However, for workloads co-scheduled on a CMP where cores share a cache, we need one more statistic. With a shared cache, multiple streams of references from multiple upper level caches compete for cache space. In such a situation, the rate of memory references from each stream to the shared cache needs to be modeled. Figure 2.6 illustrates this with a case where two copies of the same clone are co-scheduled, with the rate of memory accesses varied by changing the underlying IPC of the cores. When the IPCs of the two benchmarks are equal, as expected, the clones achieve an equal miss ratio (shown in the first three columns). However, as the IPCs of the two copies diverge, the miss ratios also diverge. Therefore, it is important to capture the rates at which memory references are generated. The rate depends on the share of memory references in the original instruction stream, the relative speed of the processor cores, and instruction level parallelism. WEST clones do not produce instruction streams other than memory references. Hence, to control the rate, the distance between successive memory references must be controlled at the processor level. To achieve that, when the original benchmark is profiled, we capture the ratio of the total number of instructions to the total number of memory references. When the WEST clone is generated, the ratio determines how many NOPs should be inserted between consecutive memory references. We refer to this metric as $\text{NOP} \text{ _per_} \text{Mref}$.

The profiling is lightweight and has a small storage overhead. It produces an output file proportional to the size of the caches. For the profiler configuration in Table 2.1, the uncompressed file size is only 233KB.
<table>
<thead>
<tr>
<th>L2 Miss Ratio (%)</th>
<th>hmm0 MR</th>
<th>hmm1 MR</th>
<th>hmm0 IPC</th>
<th>hmm1 IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.34</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.38</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.42</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.46</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.6: Co-scheduled behavior of 2 copies of hmmer in a 2M shared cache

### 2.1.3 Generation Phase

In the previous section, we reduced the application’s data cache behavior to a set of statistics. The goal of this section is to explore how a clone, that reproduces the profiled statistics, can be generated. We distinguish between three approaches based on how closely clones approximate the profiled statistics: best effort, weak statistical convergence, and strict statistical convergence.

In the best effort approach, a clone is generated in such a way that only some major features of the workload’s statistics, like a specific stride amount, are reproduced. The quality of a best effort approach clone depends on the coverage of the selected features, and can be improved by including more features. When all features of the workload’s statistics are taken into account in clone generation, the coverage reaches 100%, allowing a clone to potentially reach statistical convergence with the workload’s statistics. There are two types of statistical convergence: weak and strict. In a weak statistical convergence, the clone produces statistics that probabilistically converge to the workload’s statistics.
as the number of samples are increased. In contrast, in a strict statistical convergence, the clone produces statistics that exactly match the workload’s statistics. To illustrate the difference, consider a statistic of a workload that has a mean of 5. To achieve a weak convergence, a clone may be programmed to generate each sample independently from a statistical distribution with a mean of 5. The mean of individually generated samples may or may not be equal to 5, but over a large sample size will probabilistically converge to 5. In contrast, to achieve a strict convergence, the first sample is generated from the workload’s statistical distribution, but the distribution is adjusted before the second sample is generated, and this is repeated until the statistics match exactly. For example, if the sample size is two, and the first sample has a value of 0, then the second sample must have a value of 10, in order to force the mean of the two samples to be exactly 5.

The key benefits of strict convergence are that (1) statistical identity between the workload and its clone is guaranteed, and (2) it does not require a large sample size to achieve convergence. Hence, sample size can be scaled up or down while still guaranteeing statistical identity (subject to some constraints discussed in Section 2.3.4). Therefore, WEST adopts the strict statistical convergence approach and relies on a stochastic process that generates an access stream from various statistical distributions.

Before describing how the stochastic trace is generated, we will define some notations. For the statistics we rely on (SSD, SR, WF, and SAD), we will distinguish the original statistics collected from profiling the program with an ‘O’ subscript, versus running statistics that are manipulated by the generation algorithm. One of the running statistics captures the count of memory references that still need to occur before statistical identity is reached by the clone (denoted with a ‘C’ subscript), and another captures the cumulative distribution function derived from the count statistics (denoted with a ‘CDF’ subscript). In addition, the cache level for each of the statistics is denoted with an
$Lx$ subscript, where $x$ is the cache level. For example, $SSD_{O,L2}(i,j)$ denotes the original Set Stack Distance profile at the L2 cache for the $i^{th}$ set and $j^{th}$ stack distance position.

Initially, the input to the trace generation algorithm is the total number of accesses to be generated to different levels of caches ($A_{L1}, A_{L2}$). The value of $A_{Lx}$ can be taken from the original program profile, i.e. the number of accesses that produce the data cache behavior in the profile, or selected by users. Providing a user-specified value for $A_{Lx}$ is a significant feature of WEST, in that it allows users to scale down a long running program with a very high number of accesses to a much shorter trace or binary clone, while remaining statistically identical to the original program, subject to some constraints (Section 2.3.4).

Given a scaling factor ($scale$), the first step of the algorithm is to scale down the number of accesses to all cache levels uniformly, i.e. $A_{L1} = A_{O,L1} \times scale$ and $A_{L2} = A_{O,L2} \times scale$.

### 2.1.4 Trace Clone Generation

The algorithm to generate the stochastic address trace is described below. For bookkeeping cache state, the generator uses a simple 2-level cache simulator identical to the one used by the profiler.

1. Initialization: Various arrays are initialized using the profiled statistics. Count arrays keep track of the running count of remaining accesses to meet the different profiles. Count array values decline over time as trace generation proceeds until all values approach zero, at which point statistical identity between the clone and the
original workload is reached. The following arrays are initialized (with rounding):

\[ SSD_{C,Lx}(i,j) = A_{Lx} \times SSD_{O,Lx}(i,j) \quad \forall i \forall j \]
\[ SR_{C,Lx}(i) = A_{Lx} \times SR_{O,Lx}(i) \quad \forall i \]
\[ WF_{C,Lx}(i,j) = A_{Lx} \times WF_{O,Lx}(i,j) \quad \forall i \forall j \]

2. Warmup: In this phase, we warm up the cache hierarchy by accessing a \( S_{L2} \times W_{L2} \) blocks sequentially, where \( W_{Lx} \) and \( S_{Lx} \) denote the number of ways and sets of the cache at level \( Lx \).

3. Compute Cumulative Distribution Functions (CDF): Based on the Count Arrays, compute the various CDFs. The CDF arrays will be modified after each access is generated, to ensure strict statistical convergence regardless of the value of scale.

\[ SSD_{CDF,Lx}(i,j) = \frac{\sum_{k=1}^{j} SSD_{C,Lx}(i,k)}{\sum_{k=1}^{\infty} SSD_{C,Lx}(i,k)} \quad \forall i \forall j \]
\[ WF_{Lx}(i,j) = \frac{WF_{C,Lx}(i,j)}{SSD_{C,Lx}(i,j)} \quad \forall i \forall j \]
\[ SR_{CDF,Lx}(i) = \frac{\sum_{k=1}^{i} SR_{C,Lx}(k)}{\sum_{k=1}^{H+1} SR_{C,Lx}(k)} \quad \forall i \]
\[ SAD_{CDF,Lx}(i) = \frac{\sum_{k=1}^{i} \sum_{j=1}^{\infty} SSD_{C,Lx}(k,j)}{\sum_{k=1}^{S_{Lx}} \sum_{j=1}^{\infty} SSD_{C,Lx}(k,j)} \quad \forall i \]

4. L1 Set Selection: Generate a random number \( r \in [0,1] \). Compare \( r \) against \( SR_{CDF} \) to determine which of the recently visited sets should be visited again. More specifically, find \( \alpha_1 \) such that \( SR_{CDF,L1}(\alpha_1) \leq r < SR_{CDF,L1}(\alpha_1 + 1) \). If \( \alpha_1 \) is found, then choose the corresponding set from the history of recently accessed sets as the selected set. Otherwise (i.e., \( \alpha_1 > H \)), the selected set should not be among ones
that were recently visited. Hence, a new random number \( r \in [0, 1] \) is generated, and compared against \( SAD_{CDF,L1}(\cdot) \) to determine the set that is randomly selected, i.e. \( SAD_{CDF,L1}(\alpha_1) \leq r < SAD_{CDF,L1}(\alpha_1 + 1) \).

5. L1 Stack Position Selection: Generate a random number \( r \in [0, 1] \) and compare it to \( SSD_{CDF,L1}(\alpha_1, \cdot) \) to select a stack position \( \beta_1 \) such that \( SSD_{CDF,L1}(\alpha_1, \beta_1) \leq r < SSD_{CDF,L1}(\alpha_1, \beta_1 + 1) \). If no such \( \beta_1 \) is found, then we need to generate a miss in the L1 cache.

6. L1 R/W Selection: Generate a random number \( r \in [0, 1] \). If \( r \leq WF_{L1}(\alpha_1, \beta_1) \), the operation should be a write operation. Otherwise, it should be a read operation.

7. L1 Address Generation: If step 5 determined a cache miss, then go to step 8. To generate a hit in the L1 cache, we select the block in set \( \alpha_1 \) with the LRU stack position \( \beta_1 \) and generate an access to the block address. Based on what we determined in step 6, the operation is a read or a write. Go to step 12.

8. L2 Set Selection: The selection of a set in the L2 cache is partially determined by the set selected in the L1 cache because \( \alpha_1 \) can only map to a certain group of sets in the L2 cache. Thus, the set in the L2 cache must be selected from among the sets in this group. To do that, similar to the L1 set selection, a random number \( r \in [0, 1] \) is selected and compared to \( SAD_{CDF,L2} \) to probabilistically determine the selected set \( (\alpha_2) \). The process is repeated until the selected set is also one that the L1 cache set maps to (the repeats can be avoided with minor mathematical transformations).

9. L2 Stack Position Selection: Generate a random number \( r \in [0, 1] \) and compare it to \( SSD_{CDF,L2}(\alpha_2, \cdot) \) to select a stack position \( \beta_2 \) such that \( SSD_{CDF,L2}(\alpha_2, \beta_2) \leq \ldots \)
If no such $\beta_2$ is found, then we need to generate a miss in the L2 cache.

10. L2 Address Generation: If step 9 determined an L2 cache miss, then go to step 11.
   To generate a hit in the L2 cache, we select the block in set $\alpha_2$ with stack position $\beta_2$ and generate a read access to the block.

11. Memory Access Generation: Randomly generate a block address that maps to the set $\beta_2$ but not already present in the L2 cache. When the memory footprint is equal or larger than twice the size of the L2 cache, such an address can always be generated.

12. Adjust the Count arrays to account for the access just issued. The adjustment is by decrementing $SSD_{C,L2}(\alpha_1, \beta_1)$, $SR_{C,L1}(\alpha_1)$, and $WF_{C,L1}(\alpha_1, \beta_1)$ for the L1 cache. The same is applied to the L2 cache, in the case of an L1 cache miss. If less than $A_{L1}$ accesses have been generated, go to step 3.

To illustrate the trace generation algorithm, Figure 2.7 shows a simple example of generating a single memory reference for a cache with two sets and two ways. The addresses for blocks currently cached are shown in part (a) with the LRU stack position shown in parentheses. Assuming a user selects generation of 100 memory references, the initial values for the Count and CDF arrays, as well as Orig arrays, are shown in part (b). Suppose that the following random numbers will be generated: 0.5, 0.4, 0.9, and 0.8 from a random seed. To generate an address, the algorithm uses the first random number (0.5) and picks $SR_{CDF}$ index 2, which implies that a random set needs to be picked. Another random number (0.4) is used to index into the $SAD_{CDF}$ array and selects set 1. The next random number (0.9) uses $SSD_{CDF}(1)$ and selects stack position 2. These deter-
mine an L1 cache hit. The final random number (0.8) is compared against $WF_{CDF}(1, 2)$ to determine that the access is a read access. From part (a) in the figure, the algorithm determines that the access is a ”read to block a”. Part (c) shows how the various arrays are updated following the access. This procedure repeats until 100 addresses are generated.

![Diagram of L1 cache access process]

Figure 2.7: Trace Generation Algorithm for a single memory reference (a), contents of various arrays before address generation (b) versus after address generation (c).
2.1.5 Binary Clone Generation

The binary clone generation process follows the same algorithm as trace generation. The output trace is modified by randomly distributing $(\text{NOP}_{\text{per.Mref}} \times A_{L1})$ NOPs between the $A_{L1}$ accesses. In order to generate a C-program, the first step is to allocate a large memory area that functions as the clone’s working set. Since our goal is to mimic the data cache behavior and not main memory behavior, we set the clone’s memory footprint at two times the size of the largest cache size that we will use in design space exploration. Keeping the memory at this size further improves simulation time. Then, all the NOPs and memory accesses in the trace are converted to asm directives in the C-program. The program is then compiled using a standard C compiler to generate the clone.

2.2 Evaluation Methodology

**System Configuration.** We use a full system simulation infrastructure based on Simics [35] for our evaluation. We use “System A” with an in-order x86 processor for a wide design space exploration. The machine configuration used for profiling is shown in Table 2.1. The profiler is integrated into the Simics g-cache module and the statistics discussed in Section 2.1 are output to a file. All the benchmarks are profiled once on System A and the collected statistics are used to generate the benchmark clones offline. The binary clones are then compiled using gcc version 3.4.6. The benchmark and the binary clones are then used to explore numerous machine configurations shown in Table 2.1.

To demonstrate additional capability, we use “System B”, with an out-of-order Ultra-SPARC III+ processor modeled by Simics sample-micro-arch module. Table 2.2 describes the configuration used for profiling workloads. For System B, the profiler is integrated
into the g-cache-ooo module. The profiling and clone generation method is identical to System B with profiles collected once for each benchmark and binary clones generated offline. The benchmark and the binary clones are then tested on different OOO machine configurations shown in Table 2.2.

The purpose of the wide design space exploration is to test whether the clones can still reproduce the data cache behavior of the original benchmark when it runs on machine configurations different from the one that was used for profiling. It is a validation of whether the statistics discussed in Section 2.1 sufficiently capture the behavior of the original workload for analyzing cache performance.

Table 2.1: System Configuration for System A

<table>
<thead>
<tr>
<th></th>
<th>Profiling</th>
<th>Test &amp; Design Exploration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cores</strong></td>
<td>2 x86 cores, 2.0GHz, single issue</td>
<td></td>
</tr>
<tr>
<td><strong>L1 I$/$D$</strong></td>
<td>Private 32KB, 4 way, LRU WB w/ buffer (0 WB penalty) 2 cycles</td>
<td>Private 8<del>32KB, 1</del>4 way, LRU WT and WB w/ buffer (0 WB penalty) 2 cycles</td>
</tr>
<tr>
<td><strong>L2$</strong></td>
<td>8MB, 32 way, WB, LRU 16 cycles</td>
<td>512KB<del>8MB, 2</del>32way,WB, LRU/Rand 8~16 cycles</td>
</tr>
<tr>
<td><strong>L3$</strong></td>
<td>None</td>
<td>4<del>8MB, 16</del>32 way, WB, LRU 14-16 cycles</td>
</tr>
<tr>
<td><strong>OS</strong></td>
<td>Fedora Core 10</td>
<td></td>
</tr>
<tr>
<td><strong>Other</strong></td>
<td>4GB memory with 100 cycle access, 64B block size (L1/L2/L3), MESI</td>
<td></td>
</tr>
</tbody>
</table>

**Test Methodology.** We used 28 benchmarks from the CPU2006 suite and 6 from the BioBench suite to validate WEST (wrf is excluded due to compilation issues). The benchmark suite by itself does not have any significance except that it provides us with different workload profiles for validation. For profiling and design space exploration, the
benchmarks were fast-forwarded for 5B instructions, warmed up for 250M instructions and after clearing statistics, run for 2B instructions on System A and for 250M instructions on System B. The benchmark clones are run for one full iteration to avoid ITLB misses and instruction page faults. After that, statistics are cleared and they are run to completion. For co-scheduled workloads, the clones warm up together. Clones that complete warm-up earlier are halted until the last clone completes warm-up. The clones are then run to completion with each clone halted after it completes execution.

For clone generation, it is important to understand the role of address translation. Recall that clones simply issue loads and stores with virtual addresses. If the profile is collected before address translation occurs, then the clones can be run as-is. However, if the profile is collected after address translation, then we have to ensure that the virtual addresses that the clone issues generates the same behavior as the collected profile after address translation. This can be done by using large pages for the heap in the Operating System. Since WEST uses a small memory footprint, a large page can typically contain the entire memory footprint. So, we can guarantee that the behavior exhibited by the

---

<table>
<thead>
<tr>
<th></th>
<th>Profiling</th>
<th>Test &amp; Design Exploration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>1 UltraSPARC III+ core, 2.0GHz, 64-entry ROB, Unlimited LSQ</td>
<td>4 fetch, dispatch, execute and retire per cycle</td>
</tr>
<tr>
<td></td>
<td>4 fetch, dispatch, execute and retire per cycle</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 fetch, dispatch, execute and retire per cycle</td>
<td></td>
</tr>
<tr>
<td>L1 I$/D$</td>
<td>Private 32KB, 4 way, LRU WB w/ buffer (0 WB penalty)</td>
<td>Private 8–32KB, 1–4 way, LRU WB w/ buffer (0 WB penalty)</td>
</tr>
<tr>
<td></td>
<td>2 cycles</td>
<td>2 cycles</td>
</tr>
<tr>
<td></td>
<td>2 cycles</td>
<td>2 cycles</td>
</tr>
<tr>
<td>L2$</td>
<td>2MB, 32 way, WB, LRU</td>
<td>2MB, 32 way, WB, LRU</td>
</tr>
<tr>
<td></td>
<td>12 cycles</td>
<td>12 cycles</td>
</tr>
<tr>
<td></td>
<td>12 cycles</td>
<td>12 cycles</td>
</tr>
<tr>
<td>L3$</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>OS</td>
<td>Solaris 10</td>
<td>Solaris 10</td>
</tr>
<tr>
<td>Other</td>
<td>4GB memory with 100 cycle access, 64B block size (L1/L2), MESI</td>
<td>4GB memory with 100 cycle access, 64B block size (L1/L2), MESI</td>
</tr>
</tbody>
</table>
clones in the virtual address space is identical to the behavior exhibited in the physical address space after address translation.

Kernel transactions account for approximately 1%-2% of memory references across 1 billion memory references. Although this is amortized over such a large number of references, it can have an impact on the accuracy of WEST, especially with small traces. So, during WEST simulations, we allow the kernel transactions to access the cache but they are excluded from statistics. For the main benchmark simulations, we include the kernel transactions in the statistics to account for any system calls.

The memory footprint of the main benchmark is captured during the profiling phase but we chose to not utilize it for the evaluation. WEST only requires a memory footprint that is larger than the size of the largest cache (8MB in our evaluation) to allow generation of memory accesses. Since the profiled workloads had a footprint much larger than the last level cache size and allocation of large memory has significant simulation overhead, we chose to use a memory footprint of 16MB for our evaluation.

2.3 Results and Analysis

This section presents our evaluation results in the following order: comparison of WEST clones generated using various statistics, results for various data cache configurations for system A for single core and CMP configurations, results for various data cache configurations for system B on a single core and sensitivity study. Due to the huge number of data points (hundreds per figure, corresponding to all configurations shown in Table 2.1 and Table 2.2), all the data points across all CPU2006 and BioBench benchmarks and all cache configurations are aggregated into a single figure and sorted in ascending order based on either miss ratio or IPC.
Figure 2.8 contrasts the result of clones generated using Global Stack Distance profile (GSD clone), the most commonly used statistic in literature for analyzing temporal reuse, versus clones generated using statistics that we proposed in Section 2.1.2. For this, we vary the L1 cache from 32K/4way to 16K/4way, 8K/4way, 16K/2way, 8K/2way and 8K/1way for both GSD clones and WEST clones. We observe that the GSD clones show significant errors in miss ratio (measured as the absolute difference in miss ratios) at many data points, reaching as high as 24%. WEST clones, on the other hand, show near-zero errors across almost all cases, and only shows a single point with a relatively elevated error. This result demonstrates that as we argued in Section 2.1.2, necessary and sufficient statistics must be used to generate an accurate clone. GSD is only necessary, but insufficient for that purpose.

Figure 2.8: Comparison of miss ratio errors of GSD vs. WEST clones (Section 2.1.2) for various L1 cache configurations.
2.3.1 Results on Single Core (System A)

L1/L2 Cache Size

Figures 2.9 (a), (b), (c) show the L1 and L2 miss ratios, and IPC, obtained from the clones and the actual benchmarks, as the size of the L1 (8K/1way, 16K/2way, 32K/4way) and L2 caches (512K/2way, 1M/4way, 2M/8way, 4M/16way, 8M/32way) are changed. The figures show that the clones generate near identical results compared to those from the actual benchmarks across 510 runs. The average error in miss ratio (measured as the average of absolute differences in miss ratios) are 0.068% and 0.28% for the L1 and L2 caches, respectively. The average IPC error (measured as the average of absolute differences in IPCs) is only 0.002.

![Figure 2.9: System A L1/L2 Cache Size DSE](image-url)
Number of Sets

Figures 2.10 (a),(b) compare the L2 miss ratio and IPC of clones versus benchmarks when the number of sets in the L2 cache are varied, by fixing the cache associativity at 32 but varying the cache size from 2MB to 8MB. The average L2 miss ratio and IPC error are 0.59% and 0.0028 respectively. Only in two instances, the error is slightly elevated (9%). Changing the number of sets is challenging because when the number of sets are reduced or increased, the mapping of addresses to cache sets change, leading to changes in the SSD profile. However, as the figure shows, WEST can accurately mimic the benchmark behavior even when the number of sets in the L2 cache are changed, i.e. when the SSD in the original profile no longer matches those in the test runs.

![Figure 2.10: System A Set Change DSE](image)

Additional Cache Level

We test WEST for another challenging situation, in which we add an additional cache level. Note that the profiling and clone generation was done without taking into account the existence of another cache level. To show this, we fix the L1 cache at 32KB, and add an L3 cache. To test how the L2 cache filters accesses to the L3 cache, we vary the L2 cache size from 1MB-2MB and the L3 cache size from 4-8MB. In Figures 2.11 (a),(b)
we show the miss ratios only for those benchmarks where the L3 cache receives more 1% of the accesses. However, we show the IPC for all the runs in (c). When a clone produces very few accesses to the large L3 cache, miss ratios can show a large swing in values. Fundamentally, convergence cannot be ensured with too small a sample size (more discussion in Section 2.3.4). However, IPC is not affected by the swing in miss ratios because the number of accesses are very few. We observe errors of 0.25%, 0.83% and 0.0035 in L2 miss ratio, L3 miss ratio and IPC respectively.

**Write Policy**

We test if the clones respond to a change in the write policy at the L1 cache, by switching the write policy from write-back used during profiling, to write-through and vary the L2 cache size from 1–8MB. Figure 2.12 (a) shows an average miss ratio error of 0.236% at the L2 cache. The IPC remains accurate with an average error of 0.005.
Replacement Policy

We switch the replacement policy used in the L2 cache from LRU to Random and vary the L2 cache size from 1-4MB. Figure 2.13 (a) shows an average error of 0.28% in L2 miss ratio. The average IPC error is only 0.002. There are some cases where the error is as high as 8%. For some of these pathological cases, the random replacement policy performs better than the LRU policy and the clone does not fully capture this outcome.

2.3.2 Results for Co-schedules on CMPs (System A)

In this section, we will evaluate whether clones generated from benchmarks running stand-alone in the system, can still faithfully replicate the cache behavior when they run
co-scheduled, versus when the actual benchmarks run co-scheduled. We assume a 2-core CMP with each core having a private L1 cache and both cores sharing an L2 cache. To construct co-schedules, we categorize the 28 CPU2006 benchmarks based on their L1 miss ratio, L2 miss ratio, and IPC, when they run stand-alone (Table 2.3). We selected as many benchmarks necessary to get a good representation from each bin. Overall, we selected 15 benchmarks and mixed them in every possible combination for a total of 120 mixes. Due to space constraints, we will only show miss ratio charts in this section but we will discuss IPC results.

Table 2.3: Selection of benchmarks for evaluation of co-scheduling (Benchmarks in bold selected)

<table>
<thead>
<tr>
<th>By L1 Miss Ratio</th>
<th>By L2 Miss Ratio</th>
<th>By IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cond. Benchmark</td>
<td>Cond. Benchmark</td>
<td>Cond. Benchmark</td>
</tr>
<tr>
<td>&lt;5%</td>
<td>bzip, calculix, gcc, go, gromacs, h264, libqtm, povray, sjeng, soplex, tonto</td>
<td>&lt;1% astar, calculix, deal, gamess, gromacs, h264, hmmer, namd, perl, povray, tonto</td>
</tr>
<tr>
<td>5%-10%</td>
<td>astart, cactus, deal, gamess, milc, namd, omnetpp, perl, xalan, zeus</td>
<td>1%-10% bzip, cactus, gcc, gems, go, soplex, sphinx, xalan</td>
</tr>
<tr>
<td>&gt;10%</td>
<td>bwaves, gems, hmmer, lbm, lesile, mcf, sphinx</td>
<td>&gt;10% bwaves, ibm, lesile, libqtm, mcf, milc, omnetpp, sjeng, zeus</td>
</tr>
</tbody>
</table>

L2 Cache Size

For the 120 co-schedules, we vary the L2 cache size from 512KB, 1MB to 2MB, while keeping the L1 cache fixed at 32KB. Figure 2.14 shows the L2 miss ratios for benchmarks running on core 0 (benchmark 0) and on core 1 (benchmark 1). Out of 360 simulation runs, only results where co-scheduling resulted in one or both benchmarks suffering from an L2 miss ratio increase of over 5% are shown (excluded cases show near-identical re-
The figures show a very close match between co-scheduled clones vs. co-scheduled benchmarks. The clones correctly reflect large increases in miss ratios whenever the benchmarks show such behavior, resulting in near-identical IPCs. The average error in L2 miss ratio for benchmark 0 and benchmark 1 across all 360 simulations was 3.4% and 3.1% respectively. The average error in IPC for the benchmarks is less than 0.01.

![Figure 2.14: Impact of varying L2 cache size on L2 miss ratio: benchmark 0 (left) & benchmark 1 (right).](image-url)
Figure 2.15: Impact of varying L2 cache size on IPC: benchmark 0 (left) & benchmark 1 (right).

L1 Cache Size

For the 120 co-schedules, we vary the L1 cache size between 8KB and 16KB while keeping the L2 cache size fixed at 2MB. Although the L1 cache is private to each core, varying the L1 cache size allows us to evaluate the co-schedule behavior when the filtering effect of the L1 cache is varied. Figure 2.16 shows the L2 miss ratio for the benchmarks running on core 0 and core 1. Out of a total of 240 simulations, only when at least one benchmark in a co-schedule suffers from an increase in L2 miss ratio change of over 5%, the result is shown (excluded cases show near-identical results). Again, the results show a very close match between co-scheduled clones vs. co-scheduled benchmarks. The average error in L2 miss ratio for benchmark 0 and benchmark 1 across all 240 simulations was 2.3% and 3.3% respectively. The average error in IPC for the benchmarks is less than 0.01.

2.3.3 Results on Single Core (System B)

We will test if the clones can reproduce the data cache behavior in an out-of-order processor system. We vary the L1 cache size from 8K-32K while keeping L2 cache size
fixed at 2M. As shown in Figure 2.18, we observe that the clones are accurate with an average error of 0.71% and 0.82% for L1 miss ratio and L2 miss ratio respectively.

2.3.4 Impact of Trace Length on Convergence

One of the important parameters to consider when generating a trace is the length of the trace, because it can influence the accuracy of the simulation even when strict convergence approach is employed. As the size of the trace becomes smaller, it suffers from sample sparseness due to fewer accesses to every stack position. For example, a miss ratio of 5% cannot be replicated by 10 cache accesses; at least 20 accesses are required. A trace
Figure 2.17: Impact of varying L1 cache size on IPC: benchmark 0 (left) & benchmark 1 (right).

Figure 2.18: System B (Out-of-order) Evaluation

that is too small may be insufficient to mimic behavior at the L1 cache (L1 sparseness), and, depending on the L1 miss ratio, at the L2 cache (L2 sparseness). The inaccuracy that arises due to sparseness is unrelated to the intrinsic ability of a clone to reproduce a workload’s data cache behavior.

We will evaluate the sparseness problem in this section by focusing on the “Miss” stack position to understand the minimum length of a trace needed to achieve convergence. It is important to achieve convergence at all stack positions in all caches for accuracy. We picked two profiles; one with a very low L1 miss ratio (profile A) and one with a high L1 miss ratio (profile B) for a 32K L1 cache and generated clones with 8K, 32K,
500K and 2M memory accesses. The numbers above the columns denote the number of references for each Miss stack position averaged across all sets. As seen in Figure 2.19, the sparseness problem occurs as the number of references for the Miss stack position decreases below 10. In the case of the L1 cache, the Miss stack position sees only 1% and 5% of total references for profiles A and B respectively and the small miss ratios cause higher errors (L1 sparseness) as the trace size decreases. The L2 cache has far greater number of stack positions than the L1 cache (64×) and receives far fewer accesses due to the filtering effect of the L1 cache (L2 sparseness). This implies that more memory references are needed to achieve stochastic convergence at the L2 cache. In conclusion, it is important to maintain a trace size that is sufficiently large to achieve accuracy in all significant stack positions at all cache levels to achieve stochastic convergence.

![Figure 2.19: Trace Length Analysis](image)

(a) Prof A: L1 Miss Ratio  (b) Prof A: L2 Miss Ratio  (c) Prof B: L1 Miss Ratio  (d) Prof B: L2 Miss Ratio

2.4 Related Work

**Black Box Approach to Benchmark Cloning.** The earliest contribution to the black

46
box approach to benchmark cloning was by Bell et al. [10]. They enhanced the statistical framework proposed by Eeckhout et al. [16] by creating binaries that could be executed on a simulator or real hardware. The approach captures L1 and L2 cache miss ratios during profiling and the memory access pattern is modeled as simple strided accesses. Cloning proprietary benchmarks into synthetic clones was first proposed by Joshi et al. [30] for a single core system. Similar to Bell et al., they extract key performance attributes from the proprietary workload and clone it into a synthetic benchmark. Unlike Bell et al., the framework uses only microarchitecture-independent attributes to characterize proprietary workloads. This single core framework was also extended to capture power and thermal behavior [31, 32]. However, the technique proposed by Joshi et al. could not accurately clone the behavior of workloads like CPU2006 that have a high degree of bursty misses. Ganesan et al. [21] incorporated Memory Level Parallelism information into the clone generation model to address the bursty miss problem. The main objective of all the prior cloning work is to build a strong processor model and abstract the cache behavior by using a single dominant stride value for every static load or store instruction. In contrast, the focus of WEST is on reproducing data cache behavior, even ones with arbitrary access patterns, while assuming a simple IPC model. The instruction stream component of Joshi’s benchmark cloning is largely orthogonal (but may be complimentary) to our data cache behavior cloning.

Techniques to Reduce Simulation Time. Most techniques to reduce simulation time like statistical sampling [40, 43] and analytical modeling [13, 14, 19] do not address the classified/proprietary workload problem, hence they are not relevant to this work. However, a small number of them achieve reduced simulation time by cloning, i.e. creating a short synthetic trace from the statistical profile of the workload [17, 18, 16, 26, 27]. The statistical profile is not only insufficient to fully capture data cache behavior but also
microarchitecture-dependent in some cases [17, 18, 16, 26] and can be used only in statistical simulators. In contrast, WEST’s main goal is to solve the classified/proprietary workload problem, and the reduced simulation time is merely a by-product of the techniques used in WEST. In addition, WEST does not require separate clones for different cache configurations.

2.5 Conclusions

In this work, we have proposed a new black box workload cloning framework (WEST) for replicating data cache behavior of proprietary/confidential workloads. We have analyzed statistics that sufficiently capture arbitrary data cache behavior. We have also shown how clones can be generated to achieve strict convergence to these statistics using a stochastic method. We have also discussed how the workload clones can be used in a full system simulator or real hardware.

We have shown that WEST accurately captures miss ratios at various levels of the data cache and can respond to changes in cache size, associativity, sets, write policy, replacement policy and cache hierarchy while maintaining an average error of 0.4% for miss ratio. We have also shown that WEST accurately mimics co-scheduling behavior in CMPs.

While WEST focuses on cloning data cache behavior, its ability to clone proprietary workloads automatically, and achieve matching behavior for a wide range of cache configurations, including a shared cache architecture on a CMP, at significantly reduced simulation time, makes it a very valuable tool for computer architects.
Chapter 3

MEMST: Memory Emulation using Stochastic Traces

This chapter is organized as follows: Section 3.1 provides background on MC and DRAM. Section 3.2 describes the profiling technique and clone generation methodology. Section 3.3 describes our evaluation methodology. Section 3.4 provides the results from our evaluation and analyzes the findings. Section 3.5 describes related work. Section 3.6 concludes this work.

3.1 Background

In this section, we provide a brief introduction to DRAM based memory and MC design.

3.1.1 DRAM Organization

In modern systems, the processor is connected to a single or a group of Memory Controllers (MC) through a high speed interface. The processor initiates a cacheline request
to the MC in the event of miss in the last level cache. The MC does the cacheline transfer by bursting data across the data bus. We present two views of the DRAM attached to the MC for describing how the MC handles the data transfer.

We first present the physical view. As shown in Figure 3.1, the fundamental building block of a DRAM based memory subsystem is a 2-D array of cells called a DRAM mat. Each cell is a transistor-capacitor pair and stores 1 bit of data. When a row is selected in a DRAM mat using a row address, the entire row of bits is read out to sense amplifiers and latches called a row buffer. Using a column address, a specific bit is selected from the row buffer. The process of repeatedly reading out the entire row is expensive and the row buffer serves as a cache for subsequent column accesses within the same row. The MC in modern DDR-based systems is connected to DRAM organized as dual in-line memory modules or DIMMs. Each DIMM consists of a finite number of DRAM chips connected to the memory controller through 1 or more memory channels. Memory channels can operate independently to increase memory controller bandwidth or in lockstep to increase reliability. Each DIMM consists of 1 or more ranks. Each rank represents an independent group of DRAM chips within the DIMM. All ranks within a single DIMM share address and data buses and only one rank within a DIMM may be accessed at any time. The rank comprises of a specific number of DRAM chips and each chip contributes a specific number of bits to the data returned to the processor. For example, if the data bus width is 64 bits wide and there are 8 chips in every rank, then each chip is responsible for contributing 8 out of 64 bits. This implies that there has to be 8 DRAM mats per DRAM chip (64 DRAM mats in the rank) each providing a single bit. This group of DRAM mats within the chip that work in unison is referred to as a bank. At the DRAM chip level, when a row address is provided to the bank, the entire row from each DRAM mat is read out into the individual row buffers. The individual row buffers within each mat can be
viewed as an aggregated row buffer at the bank level. At the rank level, the same bank within all chips work in unison to generate data bus width worth of data. When a bank address is provided, the same bank within all chips is selected. Row address reads out entire rows from all DRAM mats within the selected banks from all chips. We will refer to this set of bits at the rank level as a page. The per-bank row buffers in each chip can be viewed as an aggregated per-bank row buffer at the rank level. Modern DRAM chips comprise of multiple banks for higher performance. Multiple banks increase parallelism because the banks within each rank can be accessed in parallel.

Figure 3.1: DRAM Physical Hierarchy. Additional chips for reliability not shown for simplicity.
Figure 3.2 presents a logical view which we will use for the rest of the work. The DRAM subsystem is organized in a hierarchical fashion consisting of channels, ranks, banks, rows and columns as shown in Figure 3.2. The processor provides the MC with a physical address which it decodes using an address mapping scheme. The MC uses the decoded address to select a channel, then a Dual In-line Memory Module (DIMM) within the channel, then a rank within the DIMM and then a bank within the rank. The MC then reads out an entire page (row) into the per-bank row buffer using a row address. This is called activating or opening a page. Finally, the MC provides a column address to select a particular column in the row buffer and data is returned to the processor.

Figure 3.2: DRAM Logical View.
3.1.2 Page Policy

Accessing data in DRAM-based memory starts with providing a row address to a particular bank. The process of reading out data into the row buffer by providing a row address to the bank is called Activate or "opening" the page. The act of reading destroys data in the DRAM and it has to be restored after the access is complete. The restoration can be on the critical path because at a minimum, a new row can not be activated until data for the current row has been restored. The columns in the row buffer can be then be read out from the row buffer. Once the columns are read out, the current page can be Precharged or closed.

When the row address of a new request matches the row address of the page latched in the buffer, it is a bank hit else a bank conflict. Applications with low page locality are expected to incur a high number of bank conflicts while applications with high page locality would generate more bank hits.

MCs implement a specific Page Policy which controls when a page is closed. In Open-page policy, the open page is not closed until the arrival of a request to a new page in the same bank. This implies that the current page has to be first closed before opening a new page. Open-page policy optimizes for bank hits and amortizes the cost of activates over a number of accesses to the same row. However, for a bank conflict, Open-page policy can incur the penalty of closing the page. In Close-page policy, the open page is closed immediately after the column is accessed. It optimizes for bank conflicts and attempts to hide the precharge latency in the event of a bank conflict. However, consecutive accesses to the same row will generate more activates than Open-page policy. So, Open-page policy favors applications with high page locality while Close-page favors applications with more random accesses. Adaptive Page policy attempts to
provide a balanced approach between Open-page and Close-page policy. A common implementation of such Adaptive-page policy is to close the open page after a specific period of time without waiting for a conflicting request.

### 3.1.3 Scheduling Policy

All memory requests originating from different processor cores are buffered in a FIFO queue at the memory controller. Scheduling Policy deals with how the MC schedules the requests out of order to the underlying DRAM subsystem. In the simplest First-Come-First-Serve (FCFS) implementation, the MC processes the requests in the same order as in the FIFO queue. In Open Bank First (OBF), transactions that will hit an open page are prioritized. In Read and Instruction Fetch First (RIFF), transactions that are on the critical path of the processor are given higher priority. Least-Pending and Most-Pending prioritize transactions from cores that have fewest transactions pending and most transactions pending respectively.

### 3.1.4 Address Mapping Policy

In Section 3.1.1, we discussed how the MC examines the physical address provided by the processor and orchestrates the access across multiple channels, DIMMs, Ranks and Banks. Address Mapping Scheme describes how the physical address is mapped into Channel ID, DIMM ID, Rank ID, Bank ID, Row ID and Column ID. The Address Mapping Policy is dependent on the configuration of the DRAM memory subsystem and the Page Policy.

We will use an example to describe how the DRAM memory organization determines the Address Mapping Policy. Suppose the memory subsystem supports up to 16GB of
memory across two memory channels with a maximum of 2 ranks per channel. Modern DDR3 DIMMs support up to 8 banks per rank. Suppose we use two 8GB dual rank DIMMs to reach 16GB. If the DRAM chip uses 4Gb technology, then each 8GB DIMM will have 16 chips. These 16 chips can be organized as 8 chips across two ranks with each chip contributing 8 bits. The 4Gb chip uses a 64Kx1K organization for each mat and 8 bits per chip implies 8 such mats. For such a system that uses 34 bits of physical address, it will need 1 bit for Channel ID, 1 bit for Rank ID, 3 bits for Bank ID, 16 bits for Row ID, 10 bits for Column ID and 3 bits for offset within 8B word. The page size will be 8K.

Figure 3.3 shows different address mapping schemes. Scheme 1 and 2 are optimized for Open-page policy because they map the column bits to the lower order of the address. Consecutive accesses will likely map to different columns in the same row in the same bank. Scheme 3 is optimized for Close-page because it maps the column bits to the higher order bits in the address. Consecutive accesses will map to different banks.

![Figure 3.3: Address Mapping Schemes](image-url)
3.2 MEMST Framework

The goal of our Memory EMulation using Stochastic Traces (MEMST) framework is to capture and clone the behavior of workloads in CMPs that are visible at the MC/DRAM level. The workload may be individual, multi-programmed sequential or parallel workloads. For a multi-programmed co-scheduled workload, we will clone not just the co-scheduled workload but each of the individual workloads running standalone and in a co-schedule. For parallel workloads, we will clone the overall workload and the individual threads.

We assume MEMST is used for evaluating MC/DRAM design search space, but not core and cache hierarchy design search space. To achieve that, we capture the profile of last level cache miss streams, i.e. memory references that normally would be seen by the MC. The profile must be re-collected if the cores or cache hierarchy change, or when different workloads run on the cores. However, a single profile can be used across many MC/DRAM organizations, including scheduling policy, page policy, DRAM timings, etc.

Components of the MEMST framework are shown in Figure 3.4. The input to MEMST is a profile (c) containing statistics that summarize behavior visible below the last level cache. The profile can be gathered by running either a proprietary workload or a benchmark or any other program (a) on real hardware or a simulator. Modern processors like POWER [2], ARM [1] and ARC [5] support on-chip tracing techniques that can be used to gather traces. These traces can be profiled offline. Custom profiles (b) can also be created to explore different design scenarios or for validation/debugging. The profile is fed to a unified generator (d). Using the profile, the generator can stochastically produce a trace (sequence of memory references) that can be used in trace-based DRAM simulators. The references produced are in the form of a 4-tuple (address, read/write,
cycle, core). We refer to this as the *fast mode* and the trace is referred to as a *trace clone* (e). The generator can also directly run on full system simulators and real hardware or drive trace-driven simulators. We refer to this as the *real mode* and in this mode, the generator issues *timing accurate* loads and stores. The benefit of the real mode is that there are no traces that require large storage. The real mode also enhances the portability of clones.

![Figure 3.4: MEMST Framework.](image)

### 3.2.1 Analysis of Necessary Statistics

In this section, we will examine the statistics that are necessary to generate an accurate clone that can be used in lieu of the original workload. The goal of MEMST is to faithfully clone behavior visible below the last level cache, such that the clone produces
performance metrics (row buffer miss ratio, transaction latency, memory bandwidth) and DRAM power that are largely indistinguishable from the metrics produced by the original workload. We will examine what factors affect these metrics in detail.

Row Buffer Miss Ratio

We will start by looking at an important metric of DRAM performance: row buffer miss ratio. Reproducing row buffer miss ratio accurately is very important because hits to open pages will not only incur lower latency, but also lower power.

Memory references issued by the processor have both temporal and spatial locality [9]. We examined the single dominant stride approach from [21, 24] but found that they inadequately capture spatial locality and do not capture temporal locality. We examined stack distance profile from [9], which is a powerful statistic for capturing temporal locality, but found it incapable of capturing the behavior of various DRAM organizations. This is because a stack distance profile captures temporal locality well only for a fixed address mapping, i.e. mappings where the index and tag bits of the address are fixed. In contrast, for DRAM, the bits used for indexing a row buffer vary depending on the address mapping schemes (which bits are used to index ranks, banks, channels, and rows), which in turn are consciously selected to increase memory bandwidth, exploit bank-level parallelism, etc.

To elaborate, logically, we can think of DRAM-based memory as a collection of row buffers that operate similarly to direct mapped caches. Each row buffer is indexed using a tuple of (channel ID, rank ID, bank ID). The row ID is analogous to a tag: row ID match/mismatch determines row buffer hit/miss. Finally, the column ID is similar to
the block offset in cache indexing. The number of row buffers is equal to the product of
the number of banks per rank, ranks per channel, and number of channels. Note that in
a stack distance profile, it is assumed that the index portion of a cache is unchanged. In
DRAM organization, different address mappings can use different non-contiguous bits for
index, and the bits used for channel ID, rank ID, and bank ID often move dramatically
from less significant bits to more significant bits and vice versa. We show two commonly
used address mappings in Figure 3.5. It is important to observe that not only are the
bits used for rank ID and bank ID in different locations but can also be non-contiguous.
Since the temporal locality of different significant bits vary significantly, stack distance
profile is unable to capture locality across different address mappings.

![Figure 3.5: DRAM addressing and analogy to cache addressing.](image)

In order to capture locality at the level of row buffers, we will need to use a different
profiling approach. Figure 3.7 summarizes the profiling structure that we use to achieve
that. We use a structure called Row ID Table (RIT) that resembles tag arrays in caches,
except RITs are indexed using the (channel ID, rank ID, bank ID) tuple and tagged by
the Row ID. For a CMP with $C$ cores and analyzing three different DRAM organizations,
we use a \((C + 1) \times 3\) RITs. Each column represents a DRAM organization and each row comprises of \(C\) standalone RITs (one per core) and a co-scheduled RIT. Transactions tapped from the memory bus are appropriately routed to different profiling RITs. Each RIT captures the number of hits and conflicts (or misses) to each row. Recall that our goal is to replicate row buffer miss ratio for the overall co-scheduled workload, individual threads in the co-schedule and individual threads running standalone. Thus, there is one RIT for each thread in the co-schedule running standalone and one for the entire co-scheduled workload.

At this point, we will explain the notation that will be used throughout this work. We will denote each statistic by its name and the RIT that is used for profiling this statistic will be shown in subscript following the name. We will use standalone (SA) RITs i.e. RITs that only see memory references from a single core, and co-scheduled (CO) RITs i.e RITs that see interleaved memory references from multiple cores. If the statistic is an array, then it will be described in \([\cdot]\) following the subscript. We will assume that the system has \(C\) cores where core \(c \in C\) and each RIT has \(R\) rows where row \(r \in R\).

So, if we denote a specific mapping and DRAM organization as \(o\), the profiling captures the number of hits \(H_{SA,c,o}[r]\) and the number of conflicts \(C_{SA,c,o}[r]\) in row \(r\) in SA RITs and \(H_{CO,o}[c][r]\) in row \(r\) for core \(c\) and the number of conflicts as \(C_{CO,o}[c][r]\) in row \(r\) for core \(c\) for the CO RITs. The high level view of these arrays is shown in Figure 3.7.

**Transaction Latency and Memory Bandwidth**

Transaction latency measures the time when a read transaction is queued at the MC to the time when the critical word is returned to the processor. In addition to the row buffer
Figure 3.6: Logical view of DRAM.

Figure 3.7: MEMST profiling structure consists of a stack of Row ID Tables (RIT).
miss ratio, there are a number of other factors such as DRAM timings, DRAM frequency, MC scheduling policy, MC page policy, etc. that also influence the transaction latency. One key challenge here is replicating transaction latency and memory bandwidth through MC/DRAM-architecture independent statistics.

So, we start by analyzing what factors affect these metrics. Intuitively, we know that transaction latency due to DRAM timings (tRP, tCAS, etc.), DRAM frequency and MC/chipset latency are properties of the MC/DRAM subsystem and do not change due to application behavior. However, latency due to MC scheduling policy and MC page policy are dependent on application behavior. If we use adaptive page policy as an example, the inter-arrival gap between two successive transactions to the same page determines whether the second transaction would generate a hit (if the hits are clustered close together) or would cause a page reopen (if the hits are far apart). The same rationale can be applied to a transaction accessing the same page as an earlier conflict. The first transaction serves the purpose of opening the page while the second transaction can generate a hit to the same page if it is clustered close to the first transaction such that the page timer has not expired. In the event, the second transaction arrives after the expiry of the page timer, the second transaction has to reopen the page. The gap between successive conflicts is specially important. DDRx has parameters to specifically limit the number of activates in a window of time to limit power. So, not only do we have to capture how successive transactions are spaced in time but also the behavior (hits/conflicts) of successive transactions.

To model how successive transactions are spaced in time, we capture several distributions of inter-arrival gap (in cycles) between successive requests as they arrive at the MC. Note that we capture the distribution rather than a point estimate (e.g. mean or median) so that our stochastic generation process can more accurately model bursts
of references. Furthermore, we collect separate timing distributions for four different back-to-back requests: two successive hits ($H2H$), a hit followed by a conflict ($H2C$), a conflict followed by a hit ($C2H$) and two successive conflicts ($C2C$). The reason for distinguishing these distributions is because we found that successive hits to the same page occur in quicker succession than successive conflicts, while successive conflicts or a hit followed by a conflict generally have larger inter-arrival gaps than successive hits.

Figure 3.8 illustrates the arrival of requests from two cores at the MC to a single row in a RIT. The core originating the request is shown inside the circle as a suffix to ”H” (hit) or ”C” (conflict). We build four histograms, one each for inter-arrival time of $C2C$, $H2C$, $C2H$ and $H2H$ within the profiler CO RITs. We will denote these distributions as $C2C_{CO,o}[c][r][i]$, $C2H_{CO,o}[c][r][i]$, $H2C_{CO,o}[c][r][i]$ and $H2H_{CO,o}[c][r][i]$ respectively. The index $i$ records the number of occurrences of the event ($C2C$, $C2H$, $H2C$ or $H2H$) with inter-arrival gap of $i$ cycles for core $c$. For example, in the figure, two successive hits from core 0 (H0, H0) have arrival gap of 3 and 4 time units, hence for them $H2H_{CO,o}[0][r][3]$ and $H2H_{CO,o}[0][r][4]$ are both incremented. The four histograms address timings for accesses from the same core, but do not model inter-arrival gap between accesses from different cores. So, we also track the arrival gap between accesses from two different cores $T2T_{CO,o}[c][r][i]$.

To model the behavior of successive transactions, we capture the distribution of successive hits to an open page. The statistics captured so far only capture the number of hits or conflicts which is necessary but insufficient. For example, if we record 8 hits and 2 conflicts, it is insufficient to determine whether we had 8 successive hits followed by 2 conflicts or 4 successive hits followed by a conflict, then another group of 4 successive hits and a conflict. The number of successive hits to an open page reflects how well the workload amortizes the cost of activating or opening a page. This statistic is referred to
Figure 3.8: Example of inter-arrival gap showing an interleaved stream from two cores to a single row in a RIT. Inter-arrival gap shown between successive transactions. Updates to timing distributions for core0 and core1 shown below.

as Hits per Activate (HPA) and is captured by all SA RITs. The statistic is denoted as $HPA_{SA,c,o}[r][i]$ and is a histogram where index $i$ records the number of occurrences of $i$ consecutive hits. In the example shown in Figure 3.9, we show two different workloads with each circle representing a single transaction (hit/conflict). Both workloads suffer a miss ratio of 50% but with different HPA.s. The first workload generates a single hit after all three conflicts, so $HPA_{SA,0,o}[r][1]$ is incremented three times. The second workload generates three successive hits after the first conflict, so $HPA_{SA,0,o}[r][3]$ is incremented once. The first stream amortizes the cost of each activate more uniformly than the second stream.

Figure 3.9: Example of HPA for two streams of memory references.

With $HPA$, we have modeled the hits per activate for a stream of accesses from a
single core. But, we also have to preserve the HPA for the overall workload when the threads are co-scheduled. We could have used the CO RITs to gather HPA for the co-scheduled stream. Instead of capturing the HPA for the co-scheduled workload, we capture the probability distribution of where and how often the interruptions occur between hits. We refer to this distribution as Hit Interrupt Probability ($HIP$) and denoted as $HIP_{CO,o}[c][r][i]$ where index $i$ represents the probability of a stream getting interrupted after a sequence of $i$ hits. Philosophically, $HIP$ in conjunction with standalone HPA can be used to reconstruct co-scheduled HPA, but $HIP$ better facilitates the clone generation algorithm. Note that interruptions from another core that occur between two conflicts have no effect on co-scheduled HPA. In the example shown in Figure 3.10, a standalone stream of transactions is shown on the top while the same stream is shown on the bottom when it runs in a co-schedule with the arrows showing where the stream is interrupted by transactions from other threads. Each circle represents a single transaction (hit/conflict). In the standalone stream, there are two bursts of hits, one with four hits and another with two hits. Only the first burst of hits is interrupted by another thread before the burst starts. So, 50% of the time there is an interruption from another core before the burst of hits begin ($i=0$). Hence, $HIP_{CO,o}[0][0][0]$ is set to 50%. Furthermore, the first burst is partitioned into two bursts of two hits by the interruption from another thread creating a total of three bursts of two hits each. In all three bursts, there is no interruption after a single hit, so $HIP_{CO,o}[0][0][1]$ is set to 0. There are three bursts of two hits and only one is interrupted by another thread after two hits (after the second hit in the first burst). So, there is a 33% chance of getting interrupted after a burst of two hits and $HIP_{CO,o}[0][0][2]$ is to 33%.

If we use the example in Figure 3.10, we assume that any hit following an interruption from another core changes to a conflict. In the case of parallel workloads, it is possible
that a page opened by an access from a thread is followed by an access from another thread to the same page. We refer to such hits as Symbiotic hits. In such cases, the hit following an interruption results in a hit and not a conflict. We measured such hits across a wide DRAM design space for each parallel workload running 4 and 8 threads and picked the maximum for each workload. As shown in Figure 3.11, the percentage of Symbiotic hits is almost negligible for parallel workloads. This implies that while there is data sharing in the caches, there is negligible cooperation between threads at the DRAM level. So, we assume that when transactions from threads interleave, such transactions always result in a conflict.

Memory bandwidth measures the number of transactions processed by the MC over a period of time. It measures how transactions are pipelined and transaction latencies are hidden. Again, we do not explicitly model memory bandwidth, instead relying on the arrival rate timing statistics discussed earlier.

Power
DRAM power is influenced by all the factors that affect row buffer miss ratio and transaction latency. In this section, we will look at other parameters that affect power.

So far, we have not differentiated between read and write transactions. Read and write transactions have similar mechanics at the DRAM level except during the column phase. In addition to writing data to the row buffer, the modified data is also committed to the DRAM. So, read and write transactions consume different power and it is necessary to track the read and write transactions separately. Hence, instead of tracking only hits and conflicts in different profiler RITs, we explicitly track hits and conflicts for reads and writes. We will denote the number of read hits, write hits, read conflicts and write conflicts as $RH_{SA,c,o}[r]$, $WH_{SA,c,o}[r]$, $RC_{SA,c,o}[r]$ and $WC_{SA,c,o}[r]$ respectively for the SA RITs and $RH_{CO,o}[c][r]$, $WH_{CO,o}[c][r]$, $RC_{CO,o}[c][r]$ and $WC_{CO,o}[c][r]$ respectively for the CO RITs.
Figure 3.12: Example of an interleaved stream of references from three cores. The table shows TI[$c_x$][$c_y$].

We need one final statistic to model the interaction between the memory references originating from different threads. None of the statistics discussed so far capture how a thread interacts with other threads in a co-schedule. This statistic models how often accesses from a core interrupts the stream of accesses from another core. We call this statistic Thread Interleave ($TI$) and is captured by the CO RIT. Let us denote this statistic by $TI_{CO,o}[r][c_x][c_y]$. This statistic is incremented whenever core $c_y$ follows a stream of accesses from core $c_x$ in row $r$. Figure 3.12 shows a co-scheduled stream from three cores with the transaction type (hit/conflict) and core ID shown in each circle. The table shows $TI$ for the stream of accesses. $TI_{CO,o}[0][0][1]$ is set to 1 because there is a single transaction from core 1 (transaction 3) that follows the transactions from core 0. Similarly, $TI_{CO,o}[0][1][0]$ and $TI_{CO,o}[0][1][2]$ are also incremented by one because there is one transaction from core 0 (transaction 9) and core 2 (transaction 6) that follow transactions from core 1.

Finally, instruction references from the processor are profiled as an independent stream of references, that is different from the data references originating from the same core.
The size of a profile typically varies from a few KBs to a few MBs in size depending on the number of cores and DRAM organizations. The size of the profile is independent of the number of memory references being profiled, so the statistical information in a trace containing billions of references can be compressed into a small profile.

### 3.2.2 Clone Generation

In section 2.1.2, we reduced the workload to a set of key statistics. In this section, we will describe how a clone is generated from the profiled statistics. A novel feature of MEMST is that as the clone (a parallel program) runs, threads of the program read the profiled statistics, use a stochastic process that generates memory references that conform to the statistics on the fly. Such a capability is unique and provides distinct benefits of low storage overheads (no traces need to be stored) and portability.

The memory subsystem in CMPs have parallelism at two levels. There is parallelism at the core level where each core generates memory requests independently. There is also parallelism at the row buffer level known as bank-level parallelism i.e. requests to different banks proceed in parallel fashion. However, requests from different threads to the same bank are serialized by MC scheduler. To mimic the parallelism at the core level, we generate a parallel program as the clone, where each thread generates memory references for a single core. We associate each row buffer with a lock termed baton lock. Baton locks behave similar to regular locks except they allow the current owner of the lock to specify the next owner of the lock. Only the thread holding the baton lock for a particular row buffer can generate references to that row buffer enforcing serialization within a bank. The baton lock is passed amongst threads over the course of the program. Despite using the baton lock, thread(s) run simultaneously as they own multiple baton
locks of different row buffers vs. other threads, enabling bank-level parallelism even for a single core.

![Diagram of standalone and co-scheduled transactions]

Figure 3.13: Example of two standalone streams from two cores interacting in a co-schedule.

Our goal is to mimic the standalone and co-scheduled behavior accurately. So, not only do we need to faithfully capture hits and conflicts of the entire mix, but we also need to model standalone behavior and how the standalone behavior transforms in a co-schedule. A transaction that is a conflict when running standalone and is preceded by a transaction from another core when co-scheduled is termed $TC_C$. Transactions that are a hit when running standalone and are preceded by a transaction from another core when co-scheduled are converted to a conflict. These new conflicts are termed $TC_H$. Finally, all other conflicts are termed Self-Conflicts ($SC$). Figure 3.13 shows streams from two cores standalone and co-scheduled. The original standalone stream consists of five hits and three conflicts for core 0 and one hit and two conflicts for core 1. Once
the transactions interleave in a co-schedule, transaction 3 from core 0 is converted to a
TC\_H. Transaction 5 from core 0 and transactions 1 and 3 from core 1 are now TC\_C
because they are preceded by a transaction from another core. Transactions 4 and 8 from
core 0 are SC.

Putting it all together, the generator is a parallel program with each thread generating
references for one core. Each thread generates memory references as a conflict or a burst
of hits terminated by a conflict for each row buffer. Each thread generates references
to all row buffers for which it holds the baton lock. In a co-schedule, the memory
references from the thread are subject to interference from other cores transforming
certain hits to conflicts. To mimic interference, the thread passes the baton lock for the
row buffer to another thread. The instruction references are treated as an independent
stream competing with data and instruction streams from all threads.

Compute CDFs

The first step in clone generation is to convert the statistics collected in section 2.1.2
to cumulative distribution functions (CDFs), for convenience. We will start by creating
a CDF to model standalone behavior. When running standalone, a stream consists of
a burst of hits and conflicts. CDF \( p_{HC}[c][r][i] \) represents the probability of getting a
conflict or a burst of \( i \) hits. The distribution of hits running standalone is recorded by
HPA. Every burst of hits in HPA is separated by a conflict, so we will add a conflict at
the end of every HPA. The remaining conflicts are used to generate a conflict probability
recorded in index 0 of $p_{HC}$.

$$\forall c \ \forall r \ \forall i$$

$$\text{hitsbursts}_{SA,o,c}[r] = \sum_{j=1}^{\infty} HPA_{SA,o,c}[r][j]$$

$$\text{conflict}_{SA,o,c}[r] = C_{SA,o,c}[r] - \text{hitsbursts}_{SA,o,c}[r]$$

$$\text{bursts}_{SA,o,c}[r] = \text{hitsbursts}_{SA,o,c}[r] + \text{conflict}_{SA,o,c}[r]$$

$$HC_{SA,o,c}[r][0] = \text{conflict}_{SA,o,c}[r]/\text{bursts}_{SA,o,c}[r]$$

$$HC_{SA,o,c}[r][i] = HPA_{SA,o,c}[r][i]/\text{bursts}_{SA,o,c}[r]$$

$$p_{HC}[c][r][i] = \sum_{j=0}^{i} HC_{SA,o,c}[r][j]$$

We will now generate a probability array for modeling co-scheduled behavior. It provides the likelihood of a hit or a conflict in a standalone stream being interrupted by another core. HIP records the probability of where a hit is interrupted by another core. The probability of where conflicts are interrupted is recorded by $p_{TC_C}$.

$$\forall c \ \forall r$$

$$\text{conflicts}_{CO,o,c}[r] = \sum_{j=0}^{\infty} TI_{CO,o,c}[r][c][j]$$

$$TC_{HCO,o,c}[r] = H_{SA,o,c}[r] - H_{CO,o,c}[r]$$

$$TC_{CCO,o,c}[r] = \text{conflicts}_{CO,o,c}[r] - TC_{HCO,o,c}[r]$$

$$p_{TC_{C}}[c][r] = TC_{CCO,o,c}[r]/\text{conflicts}_{CO,o,c}[c][r]$$

We will compute probability arrays for read/write ratio. Transactions that are conflicts running standalone remain conflicts when co-scheduled, so the read/write ratio for
such conflicts do not change. We use $p_{RWC}$ to represent this case. Certain hit transactions are changed to $TC_H$ when running co-scheduled. However, it is not immediately clear how many read or write transactions were converted to $TC_H$ and the ratio of read/write can be different for Hits and $TC_H$. We assign $p_{RWH}$ and $p_{RWT}$ to represent the read/write probability of hits and $TC_H$ respectively.

\[
\forall c \quad \forall r
\]
\[
p_{RWH}[c][r] = RH_{CO,o,c}[c][r]/(RH_{CO,o,c}[c][r] + WH_{CO,o,c}[c][r])
\]
\[
p_{RWC}[c][r] = RM_{SA,o,c}[r]/(RM_{SA,o,c}[r] + WM_{SA,o,c}[r])
\]
\[
p_{RWT}[c][r] = (RH_{SA,o,c}[r] - RH_{CO,o,c}[c][r])/TC_H_{CO,o,c}[c][r]
\]

We convert the timing histograms $C2C_{CO,o,c}[c][r]$, $C2H_{CO,o,c}[c][r]$, $H2C_{CO,o,c}[c][r]$, $H2H_{CO,o,c}[c][r]$ and $T2T_{CO,o,c}[c][r]$ to CDFs $p_{C2C}[c][r]$, $p_{C2H}[c][r]$, $p_{H2C}[c][r]$, $p_{H2H}[c][r]$ and $p_{T2T}[c][r]$ respectively. Finally, we compute the Next Thread probability $p_{NT}$ as follows.

\[
\forall c \quad \forall r \quad \forall i
\]
\[
p_{NT}[r][c][i] = \sum_{j=0}^{i} TI_{CO,o,c}[r][c][j]/(\sum_{j=0}^{\infty} TI_{CO,o,c}[r][c][j])
\]

To use a CDF, a random number $z \in [0, 1]$ is generated and an index $\alpha$ is selected such that $p_{CDF}[][][\alpha] \leq z < p_{CDF}[][][\alpha + 1]$.

**Generator Algorithm**
As discussed earlier, the generator is a parallel program with one thread generating memory references for each core. There is a global single-entry transaction queue, one per row buffer and the thread holding the baton lock can issue transactions to this queue. Each transaction comprises of the 4-tuple \( \text{address, read/write, core ID, issue time} \). In the fast mode, the transaction 4-tuple is output to a file while in the real mode, the transaction is issued as a load or a store at the specific issue time by the thread running on the specific core. Similar to the profiler, each thread also maintains RIT data structures to track pages in the row buffer.

Threads periodically check the issue time of the transactions in the transaction queue. If the current time equals the issue time of the transaction and the transaction belongs to the thread, the transaction is issued. Once all pending transactions are issued, threads will attempt to fill the empty transaction queue entries. Threads will try to acquire the locks for the empty transaction queue entries. If the lock acquisition is successful, threads will walk the state diagram shown in figure 3.14 to either generate one transaction or pass the baton lock to another thread. This will be done for every empty transaction queue entry. This emulates thread-level and bank-level parallelism and forces serialization at a single row buffer.

From the perspective of a single thread, it issues a sequence of hits or a conflict and periodically passes control to another thread forcing a thread conflict. Transaction generation starts at state \( \text{SC\_COMP} \) where we determine whether to generate a sequence of hits or a single conflict. Generate random number \( z \) and use \( p_{HC}^1 \) to determine \( \alpha \). If \( \alpha = 0 \), then state changes to \( \text{SC\_PEND} \) (single conflict). Otherwise, state is changed to \( \text{HIT\_PEND} \) and \( \text{hits}_{SA} = \alpha \) (sequence of \( \alpha \) hits followed by a conflict). In effect, we

\[ \text{For simplicity, we ignore the core and row subscripts for the CDFs.} \]
determine standalone behavior of threads in this state.

If state changes to SC_PEND, we have to generate a transaction that results in a conflict in the row buffer. However, a check is first made to see if the conflict is preceded by a thread conflict. Generate a random number $z$ and compare it to $p_{TC_C}$ to determine if state changes to TC_C_SCHED (schedule a thread conflict). If no thread conflict is needed, a new transaction is generated that forces a conflict in the row buffer and the state is changed to SC_COMP. In the event the state changed to TC_C_SCHED, using CDF $p_{NT}$, randomly select the next core to pass the lock and change the state to TC_C_PEND. Eventually, when the thread reacquires the lock for this row buffer, it will generate a new transaction that forces a conflict and the state is changed to SC_COMP.

If the state changes to HIT_PEND, we have to generate a sequence of $hit_{SA}$ hits. We also track the number of hits since the last thread conflict as $hit_{CO}$. A check is made to see if the hit transaction is preceded by a thread conflict using $HIP$ and $hit_{CO}$. Generate a random number $z$ and if $z \leq HIP[hit_{CO}]$, then change state to TC_H_SCHED else a new transaction that generates a row buffer hit is generated. After generating the new transaction, decrement $hit_{SA}$ and increment $hit_{CO}$. If the sequence of hits is complete i.e. $hit_{SA} = 0$, then transition state to SC_PEND to generate a conflict to terminate the burst of hits. If the state changed to TC_H_PEND, a thread conflict is forced by passing the baton lock to another thread. The next thread is chosen randomly using $p_{NT}$. When the thread reacquires the lock, it generates an access to the same page that it referenced before passing the lock. By accessing the same page, we mimic the behavior of a sequence of hits in standalone mode interrupted by a thread conflict. Again, depending on whether the sequence of bursts is complete, the state is changed to SC_PEND or HIT_PEND.

To generate a new transaction in any state, the 4-tuple transaction is constructed. To
generate a hit, we check the thread private RIT and select a new cacheline in the page.
To generate a miss, we select a new page by changing the row ID. To determine whether
the transaction is a read or a write, we generate a random number and use $p_{RWH}$ or
$p_{RWC}$ or $p_{RWT}$, depending on whether the transaction is a hit, conflict or TC_H, to
determine transaction type. The thread generating the transaction uses it’s ID as the
core ID. Finally, we use one of the five timing CDFs to determine the issue time. The
CDF is chosen based on the previous (conflict or hit) and current transaction (conflict
or hit). Transactions that occur after thread conflicts use $p_{T2T}$. A random number if
generated and using the appropriate CDF, we select a value $\beta$. The issue time is set to
last transaction time + $\beta$.

Figure 3.14: State Diagram driving address generation. States generating addresses are
highlighted.

76
3.3 Evaluation Methodology

System Configuration. We use both multi-program and parallel workloads to evaluate the MEMST framework. For the multi-program workloads, we use a full system simulation infrastructure based on Simics [35] driving a detailed MC and DRAM simulator DRAMsim [42] for trace collection. We used an 8-core out-of-order UltraSPARC III+ processor modeled by Simics sample-micro-arch module. Table 3.1 describes the reference system A used to collect traces for different workload mixes. For the parallel workload trace collection, we used the gem5 [12] simulation infrastructure [25]. The infrastructure uses an out-of-order processor model loosely based on Alpha 21264. Table 3.2 details the reference system B used for collecting traces for the parallel workloads.

The traces are profiled on a standalone version of DRAMsim. DRAMsim was enhanced with a bus model and adaptive page policy. For design space exploration, we used the stand-alone version of DRAMsim and validated a number of design points as shown in Table 3.3. All DDRx specifications are from Micron’s DRAM specifications [3]. The DDR3 power model used is based on Micron’s DDR3 power model.

Table 3.1: System A: Multi-program Workloads

<table>
<thead>
<tr>
<th>Core</th>
<th>8 UltraSPARC III+ cores, 2.0GHz, 128-entry ROB 4 fetch, dispatch, execute and retire per cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caches</td>
<td>L1: 32K, 4-way, 1 cycle. L2: 2M per core, 8-way 10 cyc. 64B Block, MESI, 8 outstanding trans.</td>
</tr>
<tr>
<td>MC</td>
<td>4 memory channels at 1600MHz, 16GB, FCFS Bus Latency=10 cyc., Chipset Latency=10 cyc. Open Page</td>
</tr>
</tbody>
</table>

Test Methodology. For the multi-program workload evaluation, we used 28 benchmarks from the CPU2006 suite, 6 from the BioBench suite and triad component from
Table 3.2: System B: Parallel Workloads

<table>
<thead>
<tr>
<th>Cores</th>
<th>8 Alpha OOO cores, 2.0GHz 8 fetch, dispatch, issue and commit per cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caches</td>
<td>L1: 32K, 2-way, 1 cycle. L2: 2M per core, 8-way 20 cyc. 64B Block, MOESI, 20 outstanding trans.</td>
</tr>
<tr>
<td>MC</td>
<td>Simple Memory, 1GB, 1 memory channel at 1600MHz</td>
</tr>
</tbody>
</table>

Table 3.3: MC and DRAM Design Space

<table>
<thead>
<tr>
<th>DIMMs</th>
<th>DDR3: 2GB-2Gb-1R, 4GB-2Gb-1R 4GB-4Gb-1R (Rev D, E), 4GB-2Gb-2R 8GB-2Gb-2R, 8GB-4Gb-1R, 8GB-4Gb-2R DDR2: 2GB-512Mb-2R, 4GB-1Gb-2R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (MHz)</td>
<td>DDR3: 1600, 1333, 800 DDR2: 800</td>
</tr>
<tr>
<td>Page Policy</td>
<td>Open, Close Adaptive timer (40, 80, 120 cyc.).</td>
</tr>
<tr>
<td>Scheduling Policy</td>
<td>First-Come-First-Serve(FCFS), Greedy Reads First, Most Pending Least Pending, Open Bank First(OBF)</td>
</tr>
<tr>
<td>Addr. Mapping</td>
<td>three mappings shown in Figure 3.3</td>
</tr>
<tr>
<td>Chipset delay</td>
<td>1, 10, 30 cycles</td>
</tr>
<tr>
<td>Bus queue</td>
<td>up to 64 entries.</td>
</tr>
<tr>
<td>Ref. Policy</td>
<td>1C:*R:*B, 1C:*R:1B, 1C:1R:1B, 1C:1R:*B</td>
</tr>
</tbody>
</table>

Stream [4] to validate MEMST (wrf is excluded due to compilation issues). After profiling the individual benchmarks, we selected a subset of benchmarks ranging from low memory bandwidth to high bandwidth. We co-scheduled the selected benchmarks in 2-core, 4-core and 8-core mixes. We excluded povray and clustalw because they generated very few memory references. The benchmark mixes are shown in Table 3.4. For trace collection, the benchmarks were fast-forwarded for 5B instructions, warmed up for 250M instructions and then run for 500M instructions or until a trace of 1M references is collected. For evaluating parallel workloads, we used the PARSEC [11] benchmark suite. We profiled the workloads in 4 and 8 thread configurations and selected a subset
of 8 benchmarks. Workloads that had very high cache hit rates (blackscholes, bodytrack, freqmire, streamcluster, swaptions) were excluded from the evaluation. For trace collection, we ran the entire parallel region of interest for the selected benchmarks using the simsmall input data set. The different benchmark suites and the simulation methodology (including simulation window) do not have any significance except that they provide us with different workload profiles for validation.

It is important to point out that we use extensive design space exploration to validate our cloning framework. For validating MEMST, we used DRAMsim because it provides a very detailed memory controller and DRAM model. However, we have not explored other tools for validating whether our profiling statistics are sufficient. Given that DRAMsim is a detailed simulator and our framework is inherently designed to be agnostic to the simulator infrastructure, we expect our framework to work well with other simulators.

Table 3.4: Benchmark Mixes

<table>
<thead>
<tr>
<th>Cores</th>
<th>Benchmarks</th>
<th>Mixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>All benchmarks excluding povray and clustalw</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>deal, hmmsearch, gcc, triad, bwaves mummer, mcf, libquantum</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>deal, hmmsearch, gcc, triad, bwaves mummer, mcf, libquantum</td>
<td>70</td>
</tr>
<tr>
<td>8</td>
<td>calculix, deal, gcc, triad, bwaves, mummer hmmsearch, xalan, mcf, libquantum</td>
<td>45</td>
</tr>
</tbody>
</table>
3.4 Results and Analysis

This section presents our evaluation for various MC and DRAM configurations. We will validate if MEMST can faithfully reproduce transaction latency, row buffer miss ratio, memory bandwidth and DRAM power of the original workload for a wide design space. Due to the large number of data points, we show the results as a scatter plot with 100s of data points. The average error $\epsilon$ is shown on each plot.

3.4.1 Design Space Exploration: Multiprogram workloads

In this section, we present our evaluation of MEMST with multi-program workloads for single-core, dual-core, quad-core and octa-core CMPs.

DRAM Organization

In this scenario, we explore various DRAM configurations by changing ranks, channels, page sizes and number of rows while keeping all other variables like memory capacity, transaction policy, etc. fixed. For this, we used all the DDR3 DIMMs in table 3.3. Figure 3.15 shows the row buffer miss ratio, transaction latency and DRAM power. We show that MEMST can accurately reproduce DRAM metrics with close to perfect correlation for close to 1400 data points.

DRAM Address Mapping

In this scenario, we evaluated three DRAM address mapping schemes for two different DRAM configurations (8x2GB and 4x4GB). Again, for over 800 data points, we can maintain very good correlation as shown in Figure 3.16.
Figure 3.15: Multi-program workload evaluation: DRAM Organization DSE

(a) Row Buffer Miss Ratio
(b) Latency
(c) Power
(d) Bandwidth

Figure 3.16: Multi-program workload evaluation: DRAM Address Mapping DSE

(a) Row Buffer Miss Ratio
(b) Latency
(c) Power
(d) Bandwidth
DRAM Frequency

For exploring how MEMST clones react to changing DDR frequency from 1600 MHz (profiled frequency) to 1333MHz and 1066MHz. Understanding how workloads react to memory frequency is important not just for DRAM/MC designers but system designers as well. Recall that we do not model frequencies explicitly instead modeling only the arrival rates. Changing DRAM frequency has an impact on latency and power and we show in Figure 3.17 that we can accurately match both metrics.

![Figure 3.17: Multi-program workload evaluation: DRAM Frequency DSE](image)

(a) Row Buffer Miss Ratio  
(b) Latency  
(c) Power  
(d) Bandwidth

Figure 3.17: Multi-program workload evaluation: DRAM Frequency DSE
Scheduling Policy

Scheduling policy is another variable that is not explicitly modeled in the black box approach. We explored six policies shown in table 3.3 for all mixes for over a 1000 data points. Note that unique profiles are not needed to model different scheduling policies. We show close to perfect correlation in Figure 3.18. We found no perceptible changes to different metrics by changing the scheduling policy for the original workload. The MEMST clones follow the same trend.
Page Policy

Page policy is a crucial design point for modern MCs as it influences both performance and power. Most modern MCs implement different page policies tailored towards different workloads. For our design space exploration, we looked at open page, close page and adaptive policy. The adaptive policy keeps the page open for a predetermined period of time $\tau$. After $\tau$, the page is closed if there are no in-flight accesses to the open page. $\tau$ is an important design parameter and we set $\tau$ to 40, 80 and 120 cycles. Recall that we neither explicitly model page policies nor capture unique profiles for each policy, instead relying on carefully chosen arrival rate metrics. As shown in Figure 3.19, we can accurately capture changes in workload behavior with change in page policies for over 850 data points.

Figure 3.19: Multi-program workload evaluation: MC Page Policy DSE
DRAM Generation

Understanding performance and power changes with new DRAM technologies is invaluable to system and DRAM designers. For this portion of analysis, we used DDR2 DIMMs instead of DDR3. We also looked at two Die revisions of the 4GB 4Gb-based DDR3 DIMM. Die revisions are short-term changes that are typically associated with lower DRAM power. We show in Figure 3.20 that we can maintain close to perfect correlation as we look at different DRAM generations.

![Figure 3.20: Multi-program workload evaluation: DRAM Generation DSE](image)

(a) Row Buffer Miss Ratio  
(b) Latency  
(c) Power  
(d) Bandwidth

Figure 3.20: Multi-program workload evaluation: DRAM Generation DSE
Bus bandwidth/Chipset latency

MC designers are also presented with design choices related to bus bandwidth to the MC. By changing the depth of the input queue to the MC, the MC bandwidth can be increased or decreased. We looked at multiple queue depths (8, 32, 64) in this work. Besides DRAM latency, MC processing also adds latency to transactions called chipset delay. MC design changes can affect this latency and impact overall application performance. We do not mandate unique profiles for modeling these changes. We varied this latency by 1, 10 and 30 clocks. For over 1000 data points, we can maintain a high degree of correlation as we change the input queue depth and chipset delay. This is shown in Figure 3.21.
Refresh Policy

Finally, we chose DRAM refresh policy from four possible options shown in Table 3.3. DRAM refresh has the potential to not only steal bandwidth from memory transactions, but also influence power. Unique profiles are not needed for capturing behavior with different refresh policies. Figure 3.22 show the effect of varying the refresh policy. Again, MEMST can maintain a high degree of correlation.

MEMST was also able to successfully clone individual workload metrics running standalone and in a co-schedule.

Figure 3.22: Multi-program workload evaluation: DRAM Refresh Policy DSE
3.4.2 Design Space Exploration: Parallel workloads

Figure 3.23 presents our evaluation of MEMST with parallel workloads. We ran the benchmarks with 4 and 8 threads while varying DRAM organization, frequency, address mapping, transaction policy, DRAM generation and refresh policy. We used the same design space parameters as used in Section 3.4.1. Across the design space comprising over 400 configurations, MEMST can maintain over 0.98 correlation for row buffer miss ratio, latency and power. For some of the profiled workloads, a few threads reach completion earlier than other threads. However, MEMST generates references continuously for all threads until completion resulting in elevated errors.

Figure 3.23: Parallel workload evaluation

(a) Row Buffer Miss Ratio  (b) Latency  (c) Power
3.4.3 Sensitivity Study: Simulation Time

The number of references may be decreased in MEMST to reduce simulation time. We chose a loose convergence approach for trace generation in MEMST. A naive loose convergence approach has the potential to generate large errors as the number of references decrease due to its inability to achieve statistical convergence. However, we designed MEMST so that all statistics achieve convergence even with a small number of references without compromising accuracy. To evaluate this feature, we ran all the workload mixes with 1/2, 1/10 and 1/20 the number of memory references as the original trace. As shown in Table 3.5, MEMST achieves very good convergence and low average error even with very few memory references.

<table>
<thead>
<tr>
<th>References</th>
<th>1/2</th>
<th>1/10</th>
<th>1/20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency ($\epsilon$)</td>
<td>1.90</td>
<td>1.95</td>
<td>1.94</td>
</tr>
<tr>
<td>Power ($\epsilon$)</td>
<td>1.68</td>
<td>1.59</td>
<td>1.42</td>
</tr>
</tbody>
</table>

3.5 Related Work

Black Box Approach to Benchmark Cloning. The foundation for cloning originated with the statistical framework proposed by Eeckhout et al. [16]. Bell et al. [10] proposed black box cloning, an extension to this statistical framework. Their method generated binary clones that could be run on full system simulators or real hardware. Joshi et al. [30] proposed cloning proprietary workloads into synthetic benchmarks using microarchitecture-independent attributes. The framework was also extended to capture
power/thermal behavior [31, 32] and Memory Level Parallelism behavior [21]. Ganesan and John [24] extended the single core framework for parallel workloads. The main objective of the prior work discussed above is to build a detailed instruction model but use a simple memory behavior model (e.g., using a single stride value for every static load or store instruction). Balakrishnan and Solihin [9] proposed WEST, a framework to capture temporal locality for cloning cache hierarchy behavior. However, as will be shown later, while some aspects of MC/DRAM can be abstracted by caches, many aspects of DRAM are inherently different from caches. MC/DRAM performance is not just sensitive to miss rates (of the row buffers), but also very sensitive to various timing parameters (e.g. time to precharge, activate, read row, read column, etc.) and internal organization (channels, ranks, banks and page sizes). Furthermore, MC/DRAM have their own design issues that are very different from caches, such as DRAM frequency, address mapping, page policy, scheduling policy, bus queue depth, and refresh policy. MEMST is designed to capture all such DRAM-specific effects through a new unique set of statistics. In addition, unlike WEST, MEMST uses a fine-grained co-scheduling generation methodology that is needed to accurately model co-scheduled behavior in DRAM, e.g. to model how accesses from different threads are interleaved at the MC and internal DRAM level. We view all cloning techniques as different tools in a designer’s toolkit, largely orthogonal in function but complimentary in achieving an overall good design. Other techniques like Power virus [22, 23] maximize power consumption for a given microarchitecture but does not clone specific workload behavior.

**Techniques to Reduce Simulation Time.** Simulation time is a major challenge in exploring design space and numerous statistical sampling [40, 43] and analytical modeling [13, 14, 19] techniques exist to solve this problem. Statistical simulation is another important technique that generates a short statistical synthetic trace from a profile which
can be run on statistical simulators [17, 18, 16, 26, 27] for culling designs very early in development. These techniques are not relevant to this work because their focus is not to solve the proprietary workload problem. The focus of MEMST is cloning workload behavior; reduced simulation time is merely a by-product of our methodology.

### 3.6 Conclusions

In this work, we have proposed a black box framework (MEMST) for cloning MC/DRAM behavior for proprietary multi-program and parallel workloads. We have proposed statistics that are necessary to clone an application for exploring MC/DRAM designs. We proposed a clone generation methodology using a stochastic method that can be used in real or fast mode to drive trace-driven or full system simulators/real hardware.

Finally, we validated our proposed framework through an extensive evaluation that included changes to DRAM organization, address mapping, DRAM frequency, page policy, scheduling policy, input bus bandwidth, chipset latency, DRAM die revision, DRAM generation and DRAM refresh policy. We showed that we can maintain a very high degree of correlation for row buffer miss ratio, transaction latency, memory bandwidth and DRAM power for over 7900 data points using single-core, dual-core, quad-core and octa-core CMPs.

The ability of MEMST to clone proprietary workloads in an automated fashion, reduced simulation cost, high portability and exceptionally high accuracy across a wide design space makes it an invaluable tool for MC, DRAM and system designers.
Chapter 4

Conclusion

It has taken me a number of years to complete this journey and it would not be complete without a few final concluding remarks that reflect on the importance of the work and look at possible future direction. In my 12 years working in the industry, I have realized the importance of characterizing and understanding workload behavior. A successful design entails not just building a high performance processor but to build one that is tailored to meet workload requirements. A workload-centric approach eventually leads to a design that is high performance, cost-effective, energy-efficient and timely. However, adopting a workload-centric approach has challenges associated with it. End users rarely share workloads with designers due to the proprietary nature of code and data. Workload cloning attempts to solve this problem. With workload cloning, end users profile their workloads and the entire workload is distilled into a set of statistics. Designers can use this profile to generate a synthetic workload and this synthetic workload can be used in lieu of the original proprietary workload.

In the CMP era, as more cores are being added to the die, there is an increasing pressure on the shared memory hierarchy (caches, memory controller and DRAM) or
"uncore" resources. This trend has pushed "uncore" design to the forefront in processor architecture. So, it is of paramount importance that we have a cloning framework that allows designers to explore the memory hierarchy design space. To this end, we have proposed two cloning frameworks, one to clone data cache behavior (WEST) and one to clone memory controller/DRAM behavior (MEMST). Both techniques are black box techniques, in that, they require little to no subject knowledge of workloads. We provide an analysis of statistics that are needed to capture the essence of an application for the memory hierarchy. We also provide clone generation techniques that use the collected statistics and produce synthetic clones. Finally, we use extensive design space exploration for caches, memory controller and DRAM to validate our frameworks. Both frameworks are highly accurate and allow designers to explore a vast design space effectively in a timely manner.

While I am concluding this work personally, I hope and anticipate that there will be further research spurred on by my work. There are other areas of cloning like prefetching, interconnects, etc. that will require my work to be extended further. There is also the possibility of having a unified cloning approach that allows in-depth design exploration of the entire system. Ultimately, the different cloning techniques are like tools in a designers’ tool box. Having a diverse set of tools will only help in developing new and exciting designs. Hopefully, my work will enable such future designs.
REFERENCES


