Accurate Cloning of the Memory Access Behavior

AMRO AWAD^{1,a)} GANESH BALAKRISHNAN^{2,b)} YIPENG WANG^{3,c)} YAN SOLIHIN^{3,d)}

Received: April 16, 2016, Released: August 3, 2016

Abstract: While customizing the memory system design or picking the most fitting design for applications is very critical, many software vendors refrain from releasing their software for several reasons. First, many applications are proprietary, hence releasing them to hardware architects or vendors is not desired. Second, applications such as defense and nuclear simulations are very sensitive, hence accessing them is very restricted. Nonetheless, customizing the hardware for such applications is still important and highly desired. Workload cloning is the technique of generating synthetic clones from the original workload. The clones mimic the memory access behavior of the original workloads, hence enable exploring the design space with high level of accuracy. In this article, we survey the state-of-art cloning techniques of the memory access behavior and their uses.

Keywords: workload cloning, memory behavior, cache hierarchy

1. Introduction

It is the goal of a computer architect to design a processor that is optimal in terms of performance, cost and power. Such a design is possible only through an in-depth understanding of the workload(s) that the design is targeted for. However, such workloads are rarely available because clients do not share their software with architects due to the proprietary or confidential nature of the code or data. Examples include weapons simulation software, trading algorithm software, and software that handles trade secrets or sensitive clients' data. The proprietary workload problem is widespread in the industry.

The current solution to the problem is for designers to use a *proxy* application which is expected to mimic the traits of the original workload. This requires clients to provide a high-level description of the algorithms used in their workloads (e.g. a highly-connected large graph algorithm) to the designers. The designers then have to manually construct a program that matches the algorithmic description. This reconstructed program, called proxy application, is then used by the designers in lieu of the original software for designing the next generation processor. An example of such approach is the generation of *proxy* applications to represent the sensitive applications used in the U.S. Department of Energy national labs [13]. **Figure 1** (a) shows an example of manual generation of a representative application, i.e., proxy application. Once the proxy application is generated, it can be used to explore the design space to pick the most appropriate design. There are major potholes with proxy programs. The proxy program is at best an educated guess at the characteristics of the

Fig. 1 A comparison between the process of generating proxy applications vs. cloning.

original software. Due to improper understanding of the underlying characteristics of the workload, details lost during verbal communication and bugs in reconstructed software, the proxy program rarely mimics the characteristics of the workload that it is expected to emulate and leads to poor design choices. It requires a huge investment of time and people with subject matter expertise for the proxy program to eventually converge with the original software.

A promising alternative is to use a *clone* application. While the end goal of the clone application is to mimic the characteristics of the original software, similar to the proxy application, the process of generating a clone is very different from the proxy application. The clone generation process extracts a statistical summary of the behavior of the client's workload through profiling, followed by the generating of a program called a clone that produces the same statistical behavior, as shown in Fig. 1 (b). Workload cloning does not suffer from the pitfalls of a proxy application. Since the orig-

¹ North Carolina State University
² Advanced Micro Devices (AMD)
³ North Carolina State University
^{a)} ajawad@ncsu.edu
^{b)} balakrig@gmail.com
^{c)} ywang50@ncsu.edu
^{d)} solihin@ncsu.edu

inal workload is profiled, it precisely captures characteristics of the original software, thus eliminating guess work and miscommunication between designers and clients. Since the clone generation process is full automated, it can reconstruct the original characteristics flawlessly.

One particular aspect of workload cloning that has no satisfactory solution is cloning of memory access behavior. Most prior cloning studies have focused on the instruction level parallelism behavior [5], [9], [11], [12], [15], [16], [17]. However, we argue that cloning is needed more urgently in memory access behavior, for several reasons. First, the cache hierarchy is becoming larger (IBM Power 7+ has 80 MB last level cache) and more complex (three levels of caches are common, four levels of caches may be deployed as well). Understanding how workloads perform in such a large and complex subsystem is crucial. Second, the growth in the number of cores puts a tremendous pressure on the memory hierarchy, shifting the bottleneck of performance from cores to the memory hierarchy. Third, due to cores sharing some parts of the memory hierarchy, proprietary workloads cannot be evaluated in isolation, but in combination with other co-running workloads. Thus, it is crucial to clone the memory access behavior of proprietary workloads accurately.

In this paper, we discuss several aspects that should be considered when cloning the memory access behavior. Later, we briefly discuss the state-of-the-art cloning techniques and their merits. The rest of the paper is organized as follows. First, we start with a quick discussion of the related work on benchmark construction and cloning in Section 2. In Section 3, we describe the cloning process. Later, on Section 4, we summarize the state-of-art memory behavior cloning techniques, their merits and advances. Later, in Sections 5, 6 and 7, we study and evaluate STM framework as an example. Finally, in Section 8, we conclude our work with suggestions and outlining the future direction for the cloning work.

2. Related Work

Synthetic Workload Design: Researchers proposed generating synthetic workloads [8], [10], [20]. The goal of their work was to set up systematic methodologies to generate artificial workloads, so that the workloads can be used to evaluate and compare the performance of different systems. Examples include Dhrystone [25] and Whetstone [7]. The synthetic workloads are not derived from particular applications, hence they do not deal with proprietary workload problem.

Workload Cloning: Workload cloning is a form of synthetic workload generation that attempts to generate workloads that replicate the behavior of the original workloads. Cloning attempts to solve the proprietary workload problem where clients are unable or unwilling to share their code with computer architects, who must design and tune architectures to suit such code. Joshi et al. [15] pioneered workload cloning where clones are generated automatically. The work was extended to capture power/thermal behavior [16], [17] and Memory Level Parallelism behavior [11]. Ganesan and John [12] extended the single core framework for parallel workloads. Another study [21] explored generating a clone as a high-level language source code, which allows archi-

itecture and compiler exploration together. The prior work discussed a detailed instruction level parallelism model (e.g., instruction mixes, dependence distance, branch predictability, etc.), but abstracted memory access behavior with a simple spatial locality model: a single dominant stride value for every static load/store. Representing memory references using single dominant strides cannot accurately replicate memory access behavior, because most loads/stores have multiple stride values.

Balakrishnan and Solihin [3] proposed collecting the temporal reuse pattern of memory accesses using the stack distance profile. The study demonstrated close match in miss rates between the generated clones compared to the original applications, across cache sizes, associativities, write policies, and number of hierarchy levels. However, the stack distance profile is inherently unable to capture spatial locality.

Another cloning study [22] used an approach that started with an original source code, and perturbed the code to obfuscate it. The approach is likely still vulnerable to reverse engineering, and hence may not fully solve the proprietary workload problem.

3. Background

Workload cloning is based on extracting statistics about inherent characteristics of the proprietary applications. Such characteristics are typically independent of the design or micro-architecture used for obtaining the profile. For instance, the temporal and spatial locality of memory accesses are microarchitecture-independent characteristics that describe the access pattern of an application. Cloning techniques capture such characteristics through a set of key characteristics that enable repeating the memory behavior in a way that enables fast and accurate architectural exploration. The cloning process consists of two major steps: profiling and clone generation. *Profiling* is the process of finding the key statistics that are sufficient to repeat the memory behavior without leaking any information about the internal functionality of the cloned workload. **Figure 2** shows the profiling process.

In the first step, the application is executed on top of a simulator or a real system to capture the key characteristics. Depending on the amount of information needed to profile, the decision of how to profile the application is made. For instance, for profiling the memory access pattern, a binary instrumentation tool or a more detailed simulator could be used to obtain the required statistics. The statistics are often probability distributions that describe the inherent memory behavior of the application. To give an illustrative example, suppose that we want to collect page-level temporal

Fig. 2 The profiling process.

Fig. 3 The clone generation process.

locality behavior, an appropriate statistics may be the probability that the same page will be reused again after accessing N other pages, with varying values of $N \geq 0$.

The collected statistical profile is a text file that can be investigated easily and checked to ensure sensitive information about the application is not leaked. The only thing that needs to be released or delivered to the hardware vendors is such the statistical profile. The next step in the cloning process is the *Clone Generation*. Generating the clone is performed by a stochastic random walk over the the statistical profile, where in each step the generator decides an event to generate based on the probability distribution expressed in the statistical profile. The resulting collection of events are wrapped into a synthetic application that resembles the architectural behavior of the cloned workload, as shown in Fig. 3.

The *clone* can be an executable application or a simple memory trace that can be used for fast and accurate architectural exploration.

4. Memory Behavior Cloning

Applications' memory behavior is one of the most important aspects that affect the whole system performance. For instance, the temporal and spatial behavior of the memory accesses can affect the efficiency of the fast on-chip hardware caches and the prefetchers. Moreover, the memory access pattern can affect the memory bus contention and the internal parallelism inside memory devices. Accordingly, cloning the memory behavior accurately can enable memory architects to apply efficient optimizations and tunes for proprietary applications. Fortunately, several black-box cloning techniques have demonstrated high level of accuracy in capturing different aspects of the memory behavior. For this paper, we will give a brief overview of the state-of-art cloning techniques: STM [2], [23], MeToo [23], [24] and MEMST [4], and describe STM in more details.

4.1 STM Framework

Spatial Temporal Memory (STM) behavior cloning is a framework to clone the spatial and temporal memory access behavior. STM extracts a set of statistics that capture both of the spatial and temporal behavior of the memory accesses. Specifically, STM introduces a novel way to combine stack reuse distance profile for capturing temporal locality with a novel stride history pattern profile that captures the spatial locality, both enabling an accurate cloning for the spatio-temporal behavior of memory accesses. Consequently, STM generated clones can be used to accurately

study the impact of cache configurations, number of levels, block sizes, prefetchers and replacement policy. Furthermore, it can be used to study the TLB behavior with different page sizes and number of entries.

4.2 MeToo Framework

The memory subsystem (memory controller, bus, and DRAM) is becoming a performance bottleneck in many computer systems and design of the memory subsystem is becoming increasingly critical in achieving the performance and power goals of the multi-core system. Memory Traffic Timing (MeT², or more pronunciation-friendly MeToo) clone is a new performance cloning technique focusing on cloning the timing behavior of the DRAM traffic. To ensure the representativeness of the timing of memory accesses, MeToo proposed a novel method to profile both instruction level behavior and cache access patterns, and generate hardware independent clones of the original applications. MeToo clones can mimic the original application's timing behavior of memory traffic very well thus can be used for memory subsystem design exploration.

4.3 MEMST Framework

While MeToo focuses on cloning timing behavior of the DRAM traffic, Memory Emulation using Stochastic Traces (MEMST) is focusing on cloning the access address patterns of the DRAM. MEMST uses a novel black box cloning approach to capture the workload statistics that are key to mimicking the workload's DRAM access pattern. Sharing the same purpose with MeToo, MEMST can be used to study various aspects of memory controller and DRAM design including DRAM organization, address mapping, refresh policies, etc.

5. Use Case: STM Framework

5.1 Overview of STM

In this section, we give a more detailed description of the STM cloning framework. The goal of STM is to gather sufficient memory access trace behavior statistics from applications and generate clones that produce memory access trace behavior that match the statistics. The memory accesses that are profiled by STM are from the stream of data accesses coming out of the processor core before reaching the L1 cache. Thus, the data access streams have not been filtered yet by any cache, and represent virtual address trace inherent to the application and instruction set architecture (ISA). STM is not concerned with the timing of these accesses. Timing is intricately linked with the microarchitecture of the processor core, the access latencies and bandwidths in various memory hierarchy levels, and their feedback loop effect on the processor pipeline. Replicating the timing effect involves many issues that are different from STM's goal of replicating memory access trace behavior.

STM uses a framework similar to West [3], as shown in Fig. 4. We initially run the binary-instrumented proprietary workload on a real hardware or non-instrumented binary on a simulator. As the binary runs, we collect statistical profiles, without collecting or storing traces from the original proprietary workload. The profiling output can be easily examined by clients so they are assured

Fig. 4 High-level view of the STM framework.

no proprietary data or information are leaked. The statistical profile is then input into the clone generator, which generates either a synthetic trace of memory references (a trace clone), or an executable (a binary clone). A trace clone can be used for input to a trace-driven simulator, while a binary clone can be used for input to a execution-driven simulator or a real system.

STM differs from West in one fundamental aspect. STM seeks to be a general solution to replicating memory access trace behavior, thus it aims to capture both temporal and spatial locality behavior, unlike West which only captures temporal locality through profiling the stack distance profile [3]. Modeling spatial locality behavior provides important new capabilities. First, West can only be used for cache design space exploration. Any cache misses are assumed to go to random addresses, including the cold misses. This assumption eliminates the clone's cache miss pattern from resembling the original workload's spatial locality. Second, many cache-related structures depend on detecting and acting upon spatial locality in both cache hits and misses. For example, a stride/sequential prefetcher works by detecting sequentiality or repetitive strides of various streams at various levels of the memory hierarchy. Using West precludes cache design exploration from including prefetchers. Third, spatial locality converts to temporal locality at a larger granularity. For example, an access to the neighboring block of a recently accessed block shows up as reuse of a super-block that is twice as large as the original block. West precludes cache design exploration from varying the cache block size, or exploring behavior at the page size such as the TLB miss rate.

5.2 Challenges in Summarizing Memory Accesses

Summarizing memory accesses into a small set of statistics is fundamentally challenging. The stream of load/store addresses stem from accesses to the high-level data structures in the source

code that are laid out in a certain way by the compiler. There is very wide variation in what data structures used by program, their sizes, and how they are accessed. By the time they manifest as load or store addresses, much of the initial information has been lost. Compressing the remaining access stream into a small set of statistics incur an additional loss of information. Thus, fundamentally, the problem of cloning memory access behavior is a *lossy* information compression problem. The challenge therefore is to choose statistics or a combination of statistics that together form the least lossy information compression while at the same time have an acceptable space overheads.

We motivate our choice of statistics through a simple example of the following memory access pattern at a cache block level: 0, 1, 2, 3, 1, 2, 3, 4, 2, 3, 4, 5, 3, 4, 5, 6, ..., as shown in Fig. 5 (top box). The access pattern shows a simple block address stream, containing both spatial and temporal locality behavior. For simplicity, we will assume that the cache is fully-associative with size of 4 cache blocks and it uses Least Recently Used (LRU) replacement policy. Therefore, the access pattern will cause the cache to miss once per 4 accesses on average; because the address pattern introduces new cache block after each 3 hits on the cache, generating a hit rate of 75% for a trace length of 100,000 accesses. Now let us consider several approaches to model the memory access pattern, and our goal is to generate 100,000 accesses that match the original hit rate of the cache.

In the first approach (Fig. 5), we consider the single dominant stride, as used in prior work [15]. The stride +1 appears much more often than the stride -2. Thus, the stride +1 is the dominant stride with an average stream length of 3. When we generate 100,000 accesses with stride +1, we will get a trace of 0, 1, 2, 3, 0, 1, 2, 3, ... This trace causes a cache hit rate of near 100%, which is inaccurate, despite the fact that stride +1 is the most dominant stride by a wide margin vs. the stride -2. Clearly, we need to capture not just one stride value.

In the second approach, we use address transition graph, where we record each block address as a node in a graph, and each edge in the graph connects a block address with its successor, and its transition probability. The space requirement to store this graph is very high, because there are a lot of distinct addresses in the virtual address space. For a 52-bit address space, if we keep a successor matrix, it requires up to $2^{52} \times 2^{52} = 2^{104}$ entries. To test the statistics, we run 100,000 Monte Carlo simulations to stochastically generate 100,000-length traces. The measured cache hit rates of the stochastic traces range from 45.8 to 47.9%, with an average of 46.8%. The resulting cache block hit rate is quite far from the hit rate of the original stream accesses, indicating its inability to capture the essential spatial/temporal behavior of the original trace.

In the third approach, we record strides instead of block addresses. Strides are often repeated much more than addresses, hence they result in very compact statistics. In the approach, we record the stride occurrence frequency on a table, thus the table records that stride +1 occurs with a 75% probability but stride -2 occurs with a 25% probability. To test the approach, we stochastically generate an address trace using the stride frequency table as a guide: if a generated random number $\in [0, 1] < 0.75$, then we

Fig. 5 Approaches for modeling memory access behavior.

generate a stride +1, otherwise we generate a stride of -2. There is no guarantee that strides +1 and -2 will be interleaved in the same order as in the original workload, but over 100,000 Monte Carlo simulations, the measured cache hit rates range from 45.8 and 47.9%, with an average of 46.8%.

In the fourth approach, we record stride transitions, rather than stride frequency, in a Markov Chain. For example, a stride -2 is always followed in the original address stream by a stride +1, which means there is a 100% probability for the edge going from node -2 to node +1. Monte Carlo simulations show across 100,000 different traces, the cache hit rates range from 66.9% to 68.9%, with an average of 67.8%. While the stride transition graph approach does not yield quite accurate results yet, it opens up a new family of related approaches, which lead us to the fifth and sixth approaches.

In the fifth and sixth approaches, the next stride probability is recorded not on the current stride value, but on a history of the last M values. We refer to this as the *stride history* or *stride pattern* approach. We can think of the fourth approach as a stride pattern approach with history depth of 1. The figure shows a history depth of 2 and 3. With a history depth of 2, Monte Carlo simulation shows traces achieving cache hit rates of 74.3–75.7% and 75.0% on average. The results are much closer to the original cache hit rate. For the history depth of 3, the cache hit rate is exactly 75.0%, matching the original block hit rate.

Overall, we have seen that modeling the spatial locality behavior is challenging because fundamentally we are dealing with a lossy information compression problem. Simple heuristics alone cannot achieve good accuracies. The stride pattern approach

Fig. 6 Profiling structures.

seems to be the most promising in capturing spatial locality behavior, and consequently temporal locality as well. Despite the simplistic example, we can observe from the figure a general trend that with the stride pattern approach, as the history depth increases, accuracy increases as well. Therefore, we adopt the stride pattern approach as a foundation for STM.

5.3 STM Statistical Profiling

In the previous section, we have discussed intuition behind the choice of using the stride pattern approach. After many experiments, we have found that a combination of stack distance profile and stride pattern table work well and better than the stride pattern table alone.

Figure 6 shows our profiling structures. On the left is a Stack Distance Probability (SDP) table showing the probability of accessing the most recently used entry (stack position 0), second

Table 1 Notations used in this paper.

Symbol	Meaning
SDC_{ij}	Stack distance count table, keeping a count of accesses at row i and column j
SDP_{ij}	Stack distance probability table, keeping probability of accessing column j over any columns in row i
N_{row}	Number of rows in the stack distance profile
N_{col}	Number of columns in the stack distance profile
$SPC_{S_M, \dots, S_1 Z_k}$	Stride pattern count table, keeping counts of transitions from past stride $S_M, \dots, S_1$ to next stride Z_k
$SPP_{S_M, \dots, S_1 Z_k}$	Stride pattern probability table, keeping the probability of transitioning from past stride $S_M, \dots, S_1$ to next stride Z_k over any values of k
M	Number of the most recent stride values in the stride pattern
N_{succ}	Number of possible successor strides
f_{SD}	Fraction of accesses that update the SD tables
f_{wr}	Fraction of accesses that are writes

most recently used entry (stack position 1), and all other stack positions. As an example, the probability for stack position 0, 1, and all others for the first set are 60%, 10%, and 30%, respectively. On the right is a Stride Pattern Probability (SPP) table showing what possible stride values (and their probabilities) follow a specific past stride pattern. This table is familiar, as we have discussed it before in Fig. 5. We will mention several notations: M represents the depth of the stride history pattern, (Z_0, F_0) denote the first stride value and probability for the specific stride history pattern, (Z_1, F_1) denote the second stride value and probability for the specific stride history pattern, etc.

Table 1 summarizes the symbols that we will use for discussion and their meanings. We gather both Stack Distance (SD) and Stride Pattern (SP) profiles. The “C” or “P” suffix are added to them to distinguish a Count statistics vs. a Probability statistics. SD profiles are kept in a table with N_{row} rows and N_{col} columns. The SP profiles are kept in a table, each entry is hash indexed using a stride pattern of the last M stride values, and keeps N_{succ} possible successor stride values.

During profiling, the SD tables are updated by keeping the tags of blocks that are most recently used, depending on the number of rows and columns of the SD tables. When there is a new address that is accessed, the address is first compared against tags in the SD tables. If there is a tag match, then the appropriate entry in the SD tables is updated appropriately, i.e. SDC_{ij} is incremented and SDP_{ij} is adjusted accordingly. If there is no tag match, the SP tables are updated, as illustrated in **Fig. 7**. The history of the last M stride value is kept (Step 1), and is given to a hash function to index the SP tables (Step 2) to access the appropriate entry in the tables. For the new address that is accessed, the new stride value is calculated by subtracting the address of the new access with the last access. It is then compared against one of the successor stride values in the tables (Step 3) to see if there is a match. If there is a match, the appropriate entry is updated by incrementing the matching stride count in the SPC table and adjusting the probability number in the SPP table. If there is no match, a new successor stride is entered into the table, and the count and probability are updated.

Fig. 7 How SPP is used during profiling and trace clone generation.

We also collect other statistics during profiling. First, we track the fraction of accesses that update the SD tables (and not the SP tables) denoted as f_{SD} (Table 1). We also collect the number of writes (or accesses due to store instructions) in order to obtain the probability of an access to be a read or a write.

There are several reasons why we keep both the SD and SP profiles. First, we found empirically that it is more accurate to keep both profiles as opposed to just the SP profiles, for a given stride pattern history depth. The SP profiles in general can be made more accurate by increasing its stride pattern history depth, but the SD profile adds accuracy without requiring a large increase in the history depth of the SP profile. The SD profiles have actually been proven to contain sufficient information for obtaining cache miss rates for different cache sizes assuming a fully/set associative cache with LRU replacement policy [14], [19]. Second, since the SD profile is incapable of capturing spatial locality, we use it only for very tight reuse distance. A prefetcher often works by detecting spatial locality pattern hence the L1 cache miss stream must contain an accurate spatial locality pattern for the prefetcher to be effective, hence we rely on the SP profile to capture any access patterns beyond ones with very tight reuse distance. Finally, the profile size is reduced significantly due to the SD profile acting as a filter to the SP profile.

5.4 STM Clone Generation

In this section, we present STM clone generation procedure. We maintain SP tables with the same configurations as used for profiling. The first step in generation is adjustment or scaling. The user starts out by determining the number of memory references desired for the clone. Suppose that the original number of memory references is N_{orig} and the desired number of memory references in the clone is N_{new} . We adjust all the SDC and SPC tables' content by multiplying each element in the tables with the scaling ratio $\frac{N_{new}}{N_{orig}}$. Note, however, that due to the Law of Large Numbers, there is a limit to scaling that beyond a certain point, the stochastic trace will be incapable of replicating the behavior of the original memory access behavior. We will evaluate the effect of scaling in Section 7.

The next step in the clone generation is to choose whether to use the SD or SP profile. To do this, we generate a random number $\rho_1 \in [0, 1]$. If $\rho_1 < f_{SD}$, we generate the next access using the SD tables, otherwise the SP tables. We then decide if the access is a read or a write by generating a random number $\rho_2 \in [0, 1]$.

If $\rho_2 < f_{wr}$, we generate a write, otherwise a read. Then, we generate another random number ρ_3 to select a row i in the SD table, and another random number ρ_4 to select a column j in the selected SD table row, all according to the statistical distribution in SDP_{ij} relative to other rows and columns in the selected row. The tag that we find in the i^{th} row and j^{th} column is the address assigned to the new access. After that, the SD tables are updated.

If $\rho_1 \geq f_{SD}$, we generate the next access using the SP tables, as illustrated in Fig. 7. To do this, we use the last M strides in the stride pattern history: $S_M, \dots, S_1$ (Step A in the figure). The stride pattern history is then used to index the SPP through a hashing function and pattern tag matching (Step B). If the appropriate entry is found, then we generate another random number ρ_5 that picks a successor stride value Z_k based on the probability distribution of all successor values in the entry (Step C). The $SPC_{S_M, \dots, S_1 | Z_k}$ is then decremented by one and $SPP_{S_M, \dots, S_1 | Z_u}$ for $u = 1, \dots, N_{succ}$ are recalculated. Then, the stochastically generated new stride value is left shifted into the stride pattern history (Step D).

The reason for decrementing the SPC and recalculating the SPP is to force *strict convergence*, where the actual stride pattern probability distribution will exactly match the original memory access' stride patterns at the end of clone generation. Strict convergence allows the trace clone to converge faster and more reliably to the original behavior compared to loose convergence. This is analogous to picking a ball from a bin that has N balls of different colors, without returning the picked ball. After N picks, the distribution of colors of the picked balls exactly matches the actual distribution of colors in the bin prior to picking. Loose convergence is analogous to picking a ball from the bin and then returning it to the bin, after each pick. Picked balls from loose convergence have a distribution that may not exactly match the original color distribution in the bin, and requires a lot more pickings to get a close match.

The final few steps (not shown in the figure) include generating a random number to select a write vs. read access, updating the SP tables to reflect the new count and probability of the new stride, calculating the new address by adding the last access address to the new stride, and updating the SD tables to reflect a new address being accessed.

This process is repeated until N_{new} accesses have been generated or until relevant entries in the SP tables are exhausted.

Finally, in case the desired output is an executable clone, STM automatically generates a C++ code which contains the memory references in the trace file. We allocate a large memory space (1 Gbyte) and ensure that all of the references are accessing that range. The memory references themselves are inlined x86-64 assembly statements, each contains move instruction from the intended address to (or from) a special register if read (or written), respectively.

5.5 Consideration for Execution Phases

The clone generation can be implemented in a way to consider execution phases of a program. The reason for this is that it is possible for some applications to dramatically change their spatial locality behavior as they transition from one execution phase

Table 2 Configurations for the system used for profiling.

Aspect	Assumption
CPU	x86-64 processor, atomic mode, 2 GHz
L1 Cache	64 B blocks, 32 KB size, 2-way, LRU
L2 Cache	64 B blocks, 512 KB size, 8-way, LRU
Main memory	2 GB
OS	Gentoo release 1.12.11.1

to another. For example, during a high spatial locality phase, a prefetcher may be very effective in eliminating cache misses, but during a low spatial locality phase, a prefetcher may not be very effective. Such dramatic changes may affect the prefetcher's ability to eliminate cache misses, and may result in the trace generated by the clone to be less accurate. For example, a long period of moderately effective prefetching may not eliminate the same number of cache misses as a period of very effective prefetching followed by a period of ineffective prefetching. To achieve that, we modify the profiling phase by having an independent stride behavior profile per fixed time interval period. In the generation phase, we use the same steps shown in the generation phase, but using the appropriate SP profile depending on the corresponding interval period.

5.6 Evaluation Methodology

For profiling and validation, we use gem5 [6], a full system simulator, to run 27 SPEC CPU2006 [1] benchmarks (all except *wrf* and *tonto* due to compilation issues) with the reference input. We fast forward each benchmark for 10 billion instructions, and then profile the execution of the next 400 million instructions. Note that STM only receives statistical profiles as input, independent of the execution length. The execution can be 400 thousand, million, billion, or even trillion instructions, and the statistical profiles collected will be of the same size, and the clone generation remains the same regardless. We chose 400 million instructions only to keep the simulation runs needed for validation manageable.

The system configuration used for collecting the profiles is as shown in **Table 2**. For evaluation and testing, we use a validated trace-driven multi-level cache simulator. Our simulator is validated by comparing its miss rates with the standard cache module provided with gem5 simulator. We profile the benchmarks using the SD tables with 128 rows and 2 columns. We use a periodic interval size of 100,000 memory references. We use stride pattern history depth of 80, and cache block size of 64 Bytes. For the TLB validation, since TLB is typically fully-associative, we use a SD profile of 1 row with 16 columns, and 4 KB page size. For both studies the clones are set to generate 200 million memory references. Later in this paper, we will discuss the reasons behind using these parameters and the impact of changing them.

6. Validation and Analysis

6.1 L1 Prefetcher

In this section we validate STM for accuracy in predicting various metrics, including the L1 cache miss rate, L2 cache miss rate, and TLB miss rate. For the validation, we use almost 400 different configurations per benchmark, while only one of the configuration (shown in Table 2) was used to gather the statistical

Fig. 8 Rank accuracy comparison between STM, West, and SDS across L1 prefetcher configurations.

Fig. 9 Rank accuracy comparison between STM, West, and SDS across L1 cache and prefetcher configurations.

profiles. For the configurations, we varied the L1 and L2 cache parameters, added a prefetcher of various configurations, and varied the TLB and page sizes.

We use two metrics for validation: *rank accuracy* and correlation coefficient/factor. Correlation factor is a measure of how two sets of data are related in term of trends, ranging from -1 (negatively related) to 1 (positively related). One weakness of correlation factor is that it does not tell us how high is good enough. Another weakness is that magnitudes are not necessarily captured well, for example a data series that is scaled by a constant from another data series have a correlation value of 1.0 , despite the difference in magnitudes. In design search exploration, computer architects care greatly about *relative performance ranking*, i.e. the result of comparing one configuration versus another, to see which configuration achieves better performance (e.g. lower miss rate). Thus, one of the most important criteria for clone accuracy is whether the substitution of the original workloads with the clones yield the same or different relative performance ranking. To achieve that, for each pair of different cache/prefetcher/TLB configurations, we compare the appropriate L1/L2/TLB miss rates. If the absolute difference in the miss rate is less than 0.1% , we categorize the relative performance ranking as *indistinguishable*. If the first configuration achieves a lower miss rate by at least 0.1% , then the first configuration is categorized as *better* in relative performance compared to the second configuration. Otherwise, it is categorized as *worse* in relative performance. Next, we compare the outcome category from the original workload vs. the outcome category from the clone. If they agree, we increment a *rank accuracy* counter. We define rank accuracy as a ratio of the rank accuracy counter and the total number of pairs. Rank accuracy captures differences in magnitudes, especially with a large number of data points.

We will now discuss each set of validation results. First, we

compare STM against West and SDS approaches. We fix the L1 cache configuration at 16 KB size, 2-way associativity, and 64-byte block size. We augment the L1 cache with a stream buffer prefetcher [18]. We vary the stream buffers by varying the number of stream buffers from 1, 2, 4, 8, ..., 256, and varying each stream buffer depth from 1, 2, 4, 8, ..., 256. We also include a configuration where no prefetcher is used. This results in 82 configurations per benchmark, and 2,214 validation points overall. This case is perhaps the most important test for the ability of STM and competing approaches such as West and SDS to capture the spatial locality behavior of memory accesses. The result for the rank accuracy is shown in Fig. 8, for all benchmarks and the average of all benchmarks.

Figure 8 shows that on average, the rank accuracies for STM, West, and SDS are 97.5%, 44.8%, and 20.5%, respectively. Despite attempting to capture spatial locality behavior, SDS is unable to replicate the spatial behavior of the original workload. West performs only slightly better. STM, on the other hand, is able to achieve a very high average rank accuracy, and consistently across benchmarks where STM strictly outperforms West and SDS. This result shows that STM is able to capture spatial locality much better than West and SDS.

6.2 L1 Cache and Prefetcher Configurations

In this section, we vary the L1 cache and L1 prefetcher configurations simultaneously. As in Section 6.1, for each cache size, we vary the number of prefetcher stream buffers from 1, 2, 4, 8, ..., 256, and vary each stream buffer depth from 1, 2, 4, 8, ..., 256. For the L1 cache, we use three size/associativity configurations: 16 KB/2-way, 32 KB/4-way, and 64 KB/8-way. This results in 246 configurations per benchmark, or 6,642 data points overall.

Figure 9 shows the rank accuracy of STM, SDS, and West

in the measure of L1 cache miss rate, across benchmarks and on average. The average rank accuracies for STM, West, and SDS are 96.0%, 65.0%, and 54.9%, respectively. STM consistently achieves a higher rank accuracy than SDS, as well as West (except for h264 where STM's rank accuracy is slightly worse). West's result is contributed by its relatively high rank accuracy due to capturing the temporal locality across L1 cache configurations, but not across prefetcher configurations. On the other hand, STM's result is contributed by h264's very long stride patterns that are not sufficiently captured with the history depth of 80. Increasing the history depth from 80 to 100 and 120 result in a substantial accuracy improvement but at the expense of larger SP tables. Therefore, West slightly achieves better rank accuracy than STM. The average correlation coefficients follow a similar trend, they are 99.5%, 62.1%, and 47.4% for STM, West, and SDS, respectively. SDS achieves a negative correlation coefficient for some benchmarks, while STM achieves > 96% correlation across all benchmarks. The average and maximum absolute error for STM is 0.17% and 1.6%.

We zoom in on the difference in L1 miss rates generated by STM vs. West and SDS on one benchmark (lbm) through a *miss rate map* in Fig. 10. Figure 10 plots all 246 points in the increasing order of original L1 cache miss rates, and overlay the clones' L1 miss rates on top of them for the corresponding data points. The figure shows that the L1 miss rates generated by West and SDS clones are uncorrelated with the original L1 miss rates. In the case of West, the clone's miss rates stay almost constant at 10%. In the case of SDS, the clone's miss rates are clustered in three values: near 0%, near 3%, and near 33%. In contrast,

STM achieves miss rates largely indistinguishable with the original L1 miss rates. This confirms that the spatial locality behavior cannot be captured by neither West's stack distance profile nor SDS's dominant stride, and requires STM's stride pattern profile for capture.

To get a better insight, Fig. 11 shows the miss rate map for 12 representative benchmarks for STM. We sort the benchmarks based on STM's rank accuracy and choose six highest and six lowest benchmarks. Two benchmarks showing the worst rank accuracies are zeus and h264. The figure shows that apart from zeus and h264, the clones' L1 miss rates are very close compared to original benchmarks' L1 miss rates. For zeus and h264, there is a systematic error affecting data points with low L1 miss rates. We have discussed earlier about h264's very long stride pattern that is insufficiently captured with history depth of $M = 80$. For zeus, increasing the history depth does not improve the rank accuracy. We suspect that zeus clone's systematic error comes from

Fig. 10 Comparison of original vs. clone L1 miss rates, for STM, West, and SDS.

Fig. 11 Comparing the original vs. clone L1 miss rate across different prefetcher and cache configurations.

Fig. 12 The impact of changing the L1 cache block size.

Fig. 13 Comparing the L2 miss rates for various L2 cache configurations. The L2 cache is non-inclusive. The L1 cache uses a write-through policy. Both L1 and L2 caches use a 64-byte block size.

our statistics not capturing spatial locality within blocks, which are interleaved with spatial locality across blocks.

6.3 L1 Cache Block Size

We test the impact of changing the L1 cache block size from 64 B (default), 128 B and 256 B, while also varying the size/associativity (32 KB/4-way, 64 KB/8-way, and 128 KB/16-way), and plot the sorted original L1 miss rates for all data points of all benchmarks, and overlay the corresponding clones' L1 miss rates in Fig. 12. This case is important test because increasing the block size turns spatial locality across neighboring blocks into temporal locality. The figure shows a close fit in general (with a small number of outliers), with the average absolute miss rate error of 0.4%, average rank accuracy of 89.1%, and correlation coefficient of 98.0%.

6.4 L2 Cache Configurations

We also vary the L2 cache sizes from 256 KB, 512 KB to 1 MB, and for each size vary the associativity from 4, 8, 16, to 32, while fixing the L1 cache at 16 KB/2-way, and plot the sorted original L2 miss rates for all data points of all benchmarks, and overlay the corresponding clones' L2 miss rates in Fig. 13. The figure shows a relatively close match of the L2 cache miss rate for the original benchmarks under different L2 cache configurations. The average rank accuracy is 88.8%, correlation coefficient is 98.5%, and the average absolute L2 miss rate error is 4.0%. The error is generally higher than L1 miss rate statistics because L2 misses are accesses that have already been filtered by the L1 and the L2 cache, hence inaccuracies have compounded at the L1 and L2 levels. However, the trends still match quite well, with the outliers contributed by just one benchmark: zeus. We repeat the experiments for write-back L1 caches, while keeping everything else unchanged. In such a case, the average rank accuracy, correla-

Fig. 14 The impact of changing the number of TLB entries and the page size.

tion coefficient, and absolute L2 miss error are 87.9%, 97.4%, and 5.9%. As with the write-through case, all benchmarks match well except for zeus which remains an outlier.

6.5 TLB number of entries and page size

Another important test for STM is whether it can capture spatial locality at even larger granularities: the page size. The profiling is performed assuming a 4 KB page size, but the actual page size is varied from 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, to 128 KB. For each page size, we also vary the fully-associative TLB size from 16, 32, 64, to 128 entries. West is incapable of capturing spatial locality behavior, so this is another new capability provided by STM, which may be useful for design space exploration of the virtual memory system. There are 24 configurations per benchmark, for a total of 648 data points, shown as sorted TLB miss rate in Fig. 14. The figure shows that a close fit: an average absolute TLB miss rate error of 0.06%, rank accuracy of 95.3%, and correlation coefficient of 99.8%.

7. Sensitivity Study

In this section, we explore the sensitivity of STM to its parameters. For all experiments, for each parameter that we vary, we include various L1 cache and prefetcher configurations. First, we vary the stride pattern history depth, from $M = 30$ to $M = 100$, and show the correlation value in Fig. 15. The figure confirms that the ability to capture spatial (and temporal) locality behavior increases with the history depth. Increasing the depth makes it possible to capture some stride patterns that are long. However, since there are few such long patterns, increasing the depth beyond $M = 80$ shows a diminishing improvement.

Second, we vary the profiling interval size to test our hypothesis that the change in execution phase may impact the overall ability of STM in capturing spatial and temporal locality behavior. Figure 16 shows the correlation coefficient values across different profiling intervals ranging from a single interval (for the entire execution), 1 million, and 100,000 memory references. The figure confirms that as the interval size is reduced, the correlation coefficient increases, suggesting that STM works more accurately in capturing the memory access behavior when it profiles and models each interval separately from others. Such results are likely caused by different phases of the program exhibiting different memory access behavior. One drawback of having small profiling interval period is the increase in the profile size. Fortunately, it seems that at 100,000 memory references, the corre-

Fig. 15 The impact of varying the stride pattern history depth.

Fig. 16 The impact of varying the profiling interval period size.

Fig. 17 The impact of trace length.

lation coefficient levels off, thus we do not need to use smaller intervals.

Third, we vary the scaling ratio to test how small we can reduce the original number of references. Since STM relies on statistical convergence, if we have many small probability values to replicate, we need a relatively large number of accesses. For example, to replicate a probability value of 12.3456%, we need at least 1 million memory reference in the clone. Furthermore, the Law of Large Number requires a stochastic process to have a sufficiently large number of samples. Therefore, there is a limit on scaling down the number of memory references in the clone, however, the limit is likely in absolute number of references, and not relative to the original number of memory references. **Figure 17** shows the impact on correlation coefficient when we vary the number of clone memory references from 200 million to 20 million to 2 million. The figure shows that we do not lose accuracy with 20 million memory references. The scaling ratio depends on the original memory references, in our case the ratio is $\frac{1}{10}$ but the ratio can be much lower if the original number of memory references is much higher.

8. Conclusion and Future Directions

In this paper, we presented the concept of memory behavior cloning, discussed its rationale and needs, and discussed the state-of-the-art example frameworks. We gave an overview of three frameworks (STM, MeToo, and MEMST), and detailed discus-

sion of STM to give a more concrete idea how memory cloning is achieved. We showed how STM clones could accurately replicate the spatial and temporal memory behavior, hence enabling the exploration of cache designs, prefetcher configurations, page sizes and TLB configurations. We believe that the cloning work has sufficient contributions that make it a promising solution for exploring the architectural design space for sensitive applications without revealing the source code.

Several frameworks have been proposed for cloning various aspects of the memory behavior [2], [4], [24]. However, integrating these efforts into one framework that can generate clones that could be used for studying various aspects is still under development.

References

- [1] Spec, spec cpu2000 and cpu2006, available from <http://www.spec.org/>
- [2] Awad, A. and Solihin, Y.: Stm: Cloning the spatial and temporal memory access behavior, *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp.237–247, IEEE (2014).
- [3] Balakrishnan, G. and Solihin, Y.: West: cloning data cache behavior using stochastic traces, *Proc. IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)*, (2012).
- [4] Balakrishnan, G. and Solihin, Y.: Memst: Cloning memory behavior using stochastic traces, *Proc. 2015 International Symposium on Memory Systems, MEMSYS '15*, pp.146–157, ACM (2015).
- [5] Bell, R.H. Jr. and John, L.K.: Improved automatic testcase synthesis for performance model validation, *Proc. 19th annual International Conference on Supercomputing (ICS)*, pp.111–120 (2005).
- [6] Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D. and Wood, D.A.: The gem5 simulator, *SIGARCH Computer Architecture News*, Vol.39, No.2, pp.1–7 (Aug. 2011).
- [7] Curnow, H.J. and Wichmann, B.A.: A synthetic benchmark, *The Computer Journal*, Vol.19, No.1, pp.43–49 (1976).
- [8] Curnow, H.J., Wichmann, B.A. and Si, T.: A synthetic benchmark, *The Computer Journal*, Vol.19, pp.43–49 (1976).
- [9] Eeckhout, L., De Bosschere, K. and Neefs, H.: Performance analysis through synthetic trace generation, *Proc. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp.1–6 (2000).
- [10] Ferrari, D.: On the foundations of artificial workload design, *Proc. 1984 ACM SIGMETRICS conference on Measurement and modeling of computer systems (SIGMETRICS)*, pp.8–14 (1984).
- [11] Ganesan, K., Jo, J. and John, L.K.: Synthesizing memory-level parallelism aware miniature clones for spec cpu2006 and imantbench workloads, *Proc. IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pp.33–44 (2010).
- [12] Ganesan, K. and John, L.K.: Automatic generation of miniaturized synthetic proxies for target applications to efficiently design multicore processors, *IEEE Trans. Computers*, Vol.99, No.1 (2013).
- [13] Heroux, M., Neely, R. and Swaminarayan, S.: Asc co-design proxy app strategy, *Los Alamos Technical Report LA-UR-13*, Vol.20460.
- [14] Hill, M.D. and Smith, A.J.: Evaluating associativity in cpu caches, *IEEE Trans. Computers*, Vol.38, No.12, pp.1612–1630 (1989).
- [15] Joshi, A., Eeckhout, L., Bell, R.H. and John, L.: Performance cloning: A technique for disseminating proprietary applications as benchmarks, *Proc. IEEE International Symposium on Workload Characterization (IISWC)*, pp.105–115 (2007).
- [16] Joshi, A., Eeckhout, L. and John, L.: The return of synthetic benchmarks, *2008 SPEC Benchmark Workshop*, pp.1–11 (2008).
- [17] Joshi, A.M., Eeckhout, L., John, L.K. and Isen, C.: Automated microprocessor stressmark generation, *Proc. IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)*, pp.229–239 (2008).
- [18] Jouppi, N.P.: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, *Proc. 17th annual International Symposium on Computer Architecture (ISCA)*, pp.364–373 (1990).
- [19] Mattson, R.L., Gecsei, J., Slutz, D.R. and Traiger, I.L.: Evaluation techniques for storage hierarchies, *IBM Systems Journal*, Vol.9, No.2, pp.78–117 (1970).

- [20] Sreenivasan, K. and Kleinman, A.J.: On the construction of a representative synthetic workload, *Comm. ACM*, Vol.17, No.3, pp.127–133 (Mar. 1974).
- [21] Van Ertvelde, L. and Eeckhout, L.: Benchmark synthesis for architecture and compiler exploration, *Proc. IEEE International Symposium on Workload Characterization (IISWC)*, pp.1–11 (2010).
- [22] Van Ertvelde, L. and Eeckhout, L.: Dispersing proprietary applications as benchmarks through code mutation, *SIGOPS Operating Systems Review*, Vol.42, No.2, pp.201–210 (Mar. 2008).
- [23] Wang, Y., Awad, A. and Solihin, Y.: Synthetic generation of clone applications, *U.S. Provisional Patent Application 62/116,120*.
- [24] Wang, Y., Balakrishnan, G. and Solihin, Y.: Metoo: Stochastic modeling of memory traffic timing behavior, *2015 International Conference on Parallel Architecture and Compilation, PACT 2015, San Francisco, CA, USA, October 18–21, 2015*, pp.457–467 (2015).
- [25] Weicker, R.P.: Dhrystone: a synthetic systems programming benchmark, *Comm. ACM*, Vol.27, No.10, pp.1013–1030 (Oct. 1984).

Amro Awad is a Ph.D. candidate in Computer Engineering at North Carolina State University. Amro's research interests include workload cloning, improving the efficiency of emerging memory technologies and enhancing hardware security.

Ganesh Balakrishnan is a Senior Member of Technical Staff at AMD since 2013. He is part of the Server Architecture and Performance group and his area of focus is memory systems and multimedia arbiter micro-architecture and modeling. He has worked on server, client and GPU designs at AMD. He was previously at IBM as a

Senior Engineer for 13 years and specialized in Server, Networking and On-chip accelerator performance. He received his Ph.D. in Computer Engineering from North Carolina State University in 2014 with publications in the area of workload analysis and cloning. He holds over 15 patents in various areas.

Yipeng Wang received his B.S. degree in 2011 and he is currently a Ph.D. student in computer engineering from North Carolina State University. His research work focuses on performance modeling and benchmark profiling. His research interests also include highly efficient memory hierarchy and hardware accelerators.

Yan Solihin is a Professor of Electrical and Computer Engineering at North Carolina State University. Currently he is also a Program Director at the National Science Foundation. He obtained his B.S. degree in computer science from Institut Teknologi Bandung in 1995, B.S. degree in Mathematics from Universitas Terbuka

Indonesia in 1995, M.A.Sc degree in computer engineering from Nanyang Technological University in 1997, and M.S. and Ph.D. degrees in computer science from the University of Illinois at Urbana-Champaign in 1999 and 2002. He is a recipient of 2010 and 2005 IBM Faculty Partnership Award, 2004 NSF Faculty Early Career Award, and 1997 AT&T Leadership Award. He is listed in the HPCA Hall of Fame. He is a senior member of the IEEE. His research interests include computer architecture, computer system modeling methods and tools. He has published more than 50 papers in performance modeling and computer architecture. He has released several software packages to the public: ACAPP — a cache performance model toolset, HeapServer — a secure heap management library, Scaltool — parallel program scalability pinpointer, and Fodex — a forensic document examination toolset. He has written two graduate-level textbooks.

(Invited by Editor-in-Chief: *Nozomu Togawa*)