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

Reena Panda[1], Xinnian Zheng[2], Andreas Gerstlauer[1] and Lizy Kurian John[1]
[1]*The University of Texas at Austin,* [2]*NVIDIA*
{*reena.panda, xzheng1*}*@utexas.edu,* {*gerstl, ljohn*}*@ece.utexas.edu*

***Abstract*—Fast and accurate design-space exploration is a critical requirement for enabling future hardware designs. However, big-data applications are often complex targets to evaluate on early performance models (e.g., simulators or RTL models) owing to their complex software-stacks, significantly long run times, system dependencies and the limited speed of performance models. To overcome the challenges in benchmarking complex big-data applications, in this paper, we propose a proxy generation methodology, CAMP that can generate miniature proxy benchmarks, which are representative of the performance of big-data applications and yet converge to results quickly without needing any complex software stack support. Prior system-level proxy generation techniques model core locality features in detail, but abstract out memory locality modeling using simple stride-based models, which results in poor cloning accuracy for most applications. CAMP accurately models both core-performance and memory locality, along with modeling the feedback loop between the two. CAMP replicates core performance by modeling the dependencies between instructions, instruction types, control-flow behavior, etc. CAMP also adds a memory locality profiling approach that captures spatial and temporal locality of applications. Finally, we propose a novel proxy replay methodology that integrates the core and memory locality models to create accurate system-level proxy benchmarks. We demonstrate that CAMP proxies can mimic the original application's performance behavior and that they can capture the performance feedback loop well. For a variety of real-world big-data applications, we show that CAMP achieves an average cloning accuracy of 89%. We believe this is a new capability that can facilitate for overall system (core and memory subsystem) design exploration.**

I. Introduction

Early computer design evaluation is performed using performance models such as execution-driven simulators or RTL-based models. Unfortunately, emerging big-data applications are often complex targets to evaluate on early performance models as running such applications requires handling their complex software layers, back-end databases, third-party libraries, which are challenging (often impossible) to support on most early performance models. Also, detailed performance models are significantly slower than real hardware that makes it difficult to analyze complete execution characteristics of these long-running applications. Furthermore, effective modeling techniques require access to either the application source code or traces. Unfortunately, end-user applications or exact traces are often inaccessible due to their proprietary nature [1]. To address these challenges, a promising solution adopted by designers/researchers is to use a miniaturized representation of the end-user workloads, called a "proxy" or "clone", which mimics the end-user workload performance.

Prior system-level proxy generation proposals [1], [2], [3] model core-level locality metrics in detail, but abstract out memory locality modeling using very simple dominant stride-based models. This results in poor cloning accuracy of the proxies, especially in applications with complex memory access patterns [4], [5]. Most big-data applications are highly data-intensive and their overall system-level performance is significantly impacted by the performance of the cache and memory hierarchy [6], [7], [8]. As a result, prior system-level performance cloning techniques are not effective to study the performance of big-data applications. Few detailed cache and memory cloning techniques [4], [5] have also been proposed. For example, spatio-temporal memory (STM) [5] tracks long history-based stride transitions in the global memory reference sequence of applications to generate miniature memory clones. Such techniques generate only a memory access trace, which can be used for cache/memory hierarchy design space exploration, but do not model any core/instruction locality behavior. In reality, the processor core configuration and the application together determine processor performance, which affect the timing of requests received in the memory system. At the same time, memory performance has a feedback loop on processor performance, which in turn affects the timing of other memory requests and the overall application performance. None of the prior cloning studies accurately model the joint performance of both core and memory subsystems and their complex interactions. As such, there is a need for system-level proxy benchmarking techniques that can model both core and memory performance accurately.

To overcome the challenges in benchmarking complex big-data applications, in this paper, we propose **CAMP**, a novel proxy generation methodology that accurately models both **C**ore performance **A**nd **M**emory locality to create miniature **P**roxy benchmarks. CAMP proxies are representative of the performance of real-world big-data applications and yet, converge to results quickly without needing any complex software-stack support. To model the core performance, we adopt existing methods for generating proxy instruction streams by capturing and modeling the dependencies between instructions (instruction-level parallelism), instruction types, control-flow behavior, etc. We add an improved memory locality profiling approach that captures both the spatial and temporal locality of applications. However, as most big-data applications typically do not have a single dominant stride/offset based access pattern, it is quite challenging to control the different dynamic execution instances of the low-level static load/store instructions in the proxy to reproduce the complex memory access patterns of the original applications using synthetic data-structure accesses in the proxy code. To address this challenge, we introduce a novel proxy modeling and replay methodology that integrates the core and memory locality models to create accurate system-level proxy benchmarks. We demonstrate that CAMP proxies can mimic the original application's core as well as memory performance behavior and that they can capture the performance feedback loop between core and memory subsystem well. For a variety of real-world database applications, we show that CAMP achieves an average cloning accuracy of $\sim$89%. We believe this is a new capability that can facilitate evaluation of overall system (core, cache and memory subsystem) design-space exploration.

II. Related Work

On the systems-side, Bell et al. [9] introduced the black-box cloning approach, which used several execution-related metrics (e.g., branch misprediction rate) to create miniature proxy benchmarks. Few other studies [10], [2] used micro-architecture independent attributes to clone core-side perfor-

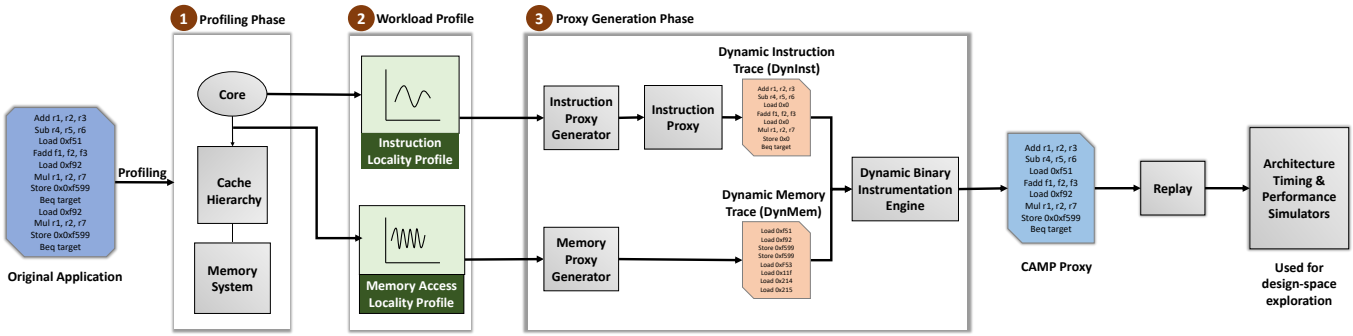


Fig. 1: CAMP's profiling and proxy generation framework

mance of proprietary applications. Such approaches use a single dominant stride-based model to characterize memory access locality and are not effective in modeling complex access patterns. As such, such proposals have been evaluated for simpler, general-purpose (e.g., SPEC CPU2006 [11]) or embedded (Implantbench [12]) benchmarks. PerfProx [3] creates proxy benchmarks for big-data applications, but it uses performance-counter based characteristics to model behavior of big-data applications. Thus, PerfProx proxies are dependent on the profiled micro-architecture and have poor cloning accuracy for other systems. CAMP creates proxy benchmarks by modeling both instruction locality and complex memory access locality using a set of micro-architecture independent metrics, and thus, enables design-space exploration of core, cache and memory hierarchy targeting big-data applications.

On the memory-side, trace-based cloning schemes [4], [5] have been explored to replicate data cache and memory performance of applications. However, they don't model other workload features, e.g., instruction dependencies, control-flow behavior, etc. Thus, they are not suitable to study system-level application performance. Other approaches such as SimPoint [13] try to reduce simulation time by identifying regions of program execution with distinct performance characteristics. However, using such techniques for database applications requires supporting complete software stacks of database applications and fast-forwarding on simulation frameworks.

III. Methodology

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

A. *Workload Profiling*

CAMP's profiling infrastructure (see ① in Figure 1) is implemented on the architecture simulator, MacSim [14] and DRAMSim2 [15] DRAM simulator. The profiler modules are developed as stand-alone modules, separate from the simulator's code. To characterize a big-data application and extract its workload statistics, we insert profiling probes into the simulation infrastructure at two points - one before the decode stage of the processor pipeline to collect the "**instruction locality profile**" and another before the data cache access ports to collect the "**memory access locality profile**". Next, we will discuss the different metrics corresponding to the instruction and memory locality profiles.

1) *Instruction Locality Parameters*

a. Basic-block features and instruction footprint - CAMP identifies the number of dominant static basic blocks in the original big-data application, which constitute a fixed threshold (empirically, set to 90% in our case) of the big-data application's total dynamic basic-block count. The number of basic blocks instantiated in the proxy benchmark is set to the number of dominant basic blocks identified in the original application. A lower threshold value can lead to a higher degree of miniaturization but at the expense of a loss in cloning accuracy. Next, CAMP tracks the average basic block size of the dominant basic blocks and transition probabilities between pairs of basic blocks. Average basic block size is an important metric because it determines the average number of instructions that can be executed in the program sequence without executing any control instructions.

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

c. Control flow behavior - Another important metric that affects big-data application performance is its control flow behavior. Difficult-to-predict branches lead to poor branch predictor performance, which causes higher number of wrong-path executions and degrades system performance. Prior research work [10], [9], [2] have shown that an application's branch misprediction rate is highly correlated with the transition frequency of the branch instructions [16]. Branch transition rate measures how often a branch transitions between its taken and not-taken paths and is an indicator of the overall branch predictability. CAMP captures the transition rate of the branch instructions in the big-data applications and bins them into eight buckets, where each bucket represents the fraction of branches with a transition rate ranging from 0-100%. To model a certain branch transition rate in the proxy, each branch instruction is assigned a transition frequency to satisfy the target branch transition rate.

d. Instruction-level parallelism - Next, CAMP captures

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

e. System activity - Many emerging, big-data applications spend a significant fraction of their execution time executing operating system code [6], [7]. To model the impact of high system activity, CAMP tracks the fraction of executed user-mode and kernel instructions in the big-data applications during profiling. Next, CAMP adds the target fraction of system calls into the proxy benchmark during proxy generation.

Table I summarizes the different instruction locality features captured by CAMP.

*2) **Memory Locality Parameters***

As discussed, prior system-level proxy benchmarking techniques use a very simple model to capture memory access locality. They model locality of individual load/store instructions in the original application based on a single dominant stride value. Although such an approach can work for small loop-based programs (e.g., array-based accesses), the memory instructions in most big-data applications have quite random, complex access patterns which cannot be captured by a single stride alone [6]. For example, join operations using hash tables, key-value stores and complex structures such b-trees do not lend themselves well to dominant strides as a representative model [7]. In this section, we will discuss how CAMP addresses the need for a more representative memory model.

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

To accurately model the read requests generated by memory instructions run by the core (type (a)), we first capture a per cache-set stack distance distribution (SSD) profile [17] for a baseline L1 cache (16KB, 2-way). The SSD profile captures the fraction of memory references to the different LRU stack positions (stack position 0 represents the most recently used block, stack position 1 represents the second most recently used block, etc.) within every cache set of the baseline L1 cache. For example, SSD_{ij} represents the probability for an access to fall in the i^{th} set at the j^{th} LRU stack position. Using SSDs helps to capture the temporal locality of memory access streams. For the accesses that miss in the L1 cache, we capture their spatial locality patterns by learning global stride transitions in a stride history table (SHT). A stride is defined as the difference between addresses of two consecutive memory accesses that miss in the profiled L1 cache. Each SHT entry records a history of past consecutive stride values (length is based on the history depth), and the next strides that followed the stride history in the past, along with each next stride's rate

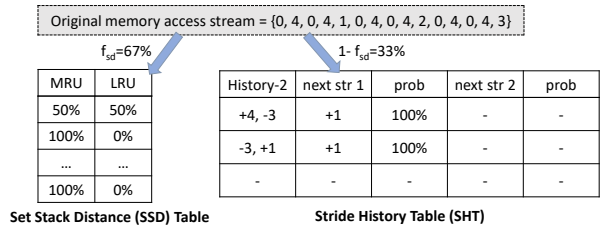


Fig. 2: STM-based memory locality profiling

of occurrence. We also collect another statistic, f_{SD}, which records the fraction of memory accesses that hit in the SSD table. Figure 2 illustrates a simplified view of these profiling structures. Note that these statistics are the only ones, which are similar to the statistics used in STM [5], all the remaining statistics are unique to CAMP.

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

However, we found that STM's statistics are not sufficient to deal with write-back request traffic in the memory system. STM collects a metric, ρ_w, which records the fraction of write accesses in the original program. But the number of write-backs is not determined by the fraction of write accesses. Rather, it depends on the number of dirty blocks in the cache hierarchy and this is not captured by STM's write statistics. Figure 3 shows an example case that leads to different number of write-backs in STM versus the original program. In the original program, 50% of the accesses are stores, but the stores occur to the same cache block, resulting in one dirty cache block. As STM does not capture the fraction of writes to clean or dirty cachelines, it can generate two writes to different blocks, resulting in two dirty blocks and two future write-back requests. In order to capture this effect, in CAMP, we record the number of writes to clean and dirty blocks. When a clean block receives a write request, it becomes dirty and any subsequent read/write operations on the same block do not impact its status. Based on the counts aggregated during the profiling phase, we compute two probabilities, W_c and W_d, which represent the probability of writing to a clean or dirty block, respectively. During proxy synthesis, we select the request type (load or store) based on the clean or dirty state of the generated address and the probabilities (W_c and W_d).

B. *Proxy Generation and Modeling*

Next, we will describe CAMP's proxy generation process (see ③ in Figure 1). Table I summarizes the locality metrics.

Algorithm 1 shows how the "memory proxy generator" leverages the memory locality profiles to create a dynamic

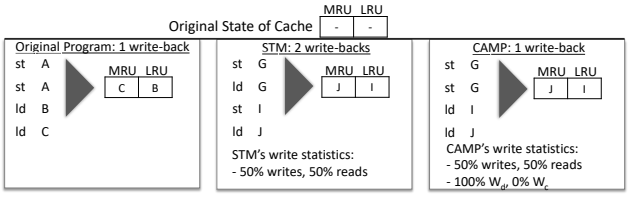


Fig. 3: Profiling for write-back requests

TABLE I: Profiled statistics

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

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

Next, the "instruction proxy generator" leverages the instruction locality metrics to create an instruction proxy (see Algorithm 2). First, CAMP populates each basic block in the proxy with an appropriate number (satisfying the mean and standard deviation of the target B_{size}) and type (satisfying f_{imix}) of instructions. The last instruction of every basic block is instantiated as a conditional branch instruction. Next, each instruction is assigned a dependency distance (i.e., a prior instruction that generates its operands) to satisfy the dependency distance criterion (line 7). As memory instructions in most big-data applications typically do not have a fixed stride/offset, it is very challenging to control the different dynamic execution instances of the low-level, static load/store instructions in the instruction proxy to dynamically produce the same dynamic memory access sequence produced by Algorithm 1 (*DynMem*). To achieve this, we temporarily assign all the memory instructions in the instruction proxy to have a zero stride with respect to a baseline array (line 8). After instruction proxy generation completes, CAMP uses a binary instrumentation engine to integrate the *DynMem* trace into the instruction proxy, as we will discuss in the next paragraph. Next, system calls are injected (or not) into the basic block based on the target system-call frequency (line 9). An x86 test operation is inserted before every branch to set the condition codes that control the branch's outcome. The test instruction's operand is chosen to control the transition frequency of the branch instruction (line 10). The above steps are repeated till the target number of basic blocks (B) are generated. Finally, architectural registers are assigned to each instruction to satisfy the identified dependencies. The instruction proxy generator generates C-language based proxies with embedded x86-based assembly instructions using the *asm* construct. The proxy instructions are nested under a two-level loop where the loop iterations control the number of dynamic instructions.

Next, the above static instruction proxy is compiled and

Algorithm 1 Dynamic memory proxy generation algorithm

1: **Output:** $DynMem[] = \{(ADDR_1, RW_1), \ldots, (ADDR_N, RW_N)\}$;
2: **for** $n = 1,\ldots,N$ **do**
3: Sample $f_n \in \{0,\ldots,100\}$;
4: **if** f_n ¡ f_{SD} **then**
5: Use S_i to choose set, SSD_{ij} to choose LRU dist. position;
6: Choose $ADDR_n$ based on chosen set and stack position;
7: **else**
8: Sample stride S_n from SHT based on $LAST\_STR$;
9: $ADDR_n = LAST\_ADDR + S_n$;
10: $LAST\_STR$.push_back(S_n); $LAST\_ADDR = ADDR_n$;
11: **end if**
12: **if** $ADDR_n \in DirtyBlocks$ **then**
13: Sample W_d and ρ_w to assign RW_n;
14: **else**
15: Sample W_c and (1-ρ_w) to assign RW_n;
16: **end if**
17: $DirtyBlocks.add(ADDR_n)$ if $RW_n = Write$;
18: **end for**

Algorithm 2 Instruction proxy synthesis algorithm

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

profiled using a binary instrumentation tool (e.g., PIN) to generate a dynamic instruction stream (*DynInst*) of the same. An example format of *DynInst* is shown in Figure 4. Next, we propose and implement a dynamic binary instrumentation engine for integrating the *DynMem* and *DynInst* profiles to create the unified CAMP proxies, capturing both instruction and memory access behavior of the original big-data applications. For every dynamic execution instance of the load/store instructions in the *DynInst* sequence, the instrumentation engine overrides the temporary address assigned to the instruction during the instruction proxy generation time with the next address from the *DynMem* sequence in a serialized fashion. The instrumented instruction and memory stream forms the CAMP proxy. To evaluate the CAMP proxies on simulators, we implement a replay engine that interfaces with the trace/binary reading logic of the architectural performance or power simulator and feeds the simulator with the unified CAMP dynamic instruction and memory sequences. Most architectural simulators (e.g., SniperSim [18], Macsim [14], Ramulator [19]) support well-defined dynamic trace driven execution modes and CAMP proxies could be easily integrated into such frameworks by modifying the replay engine.

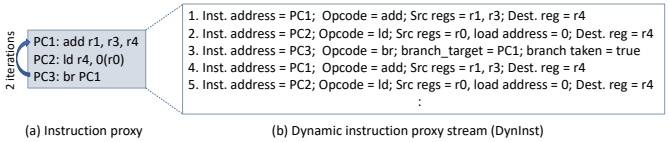
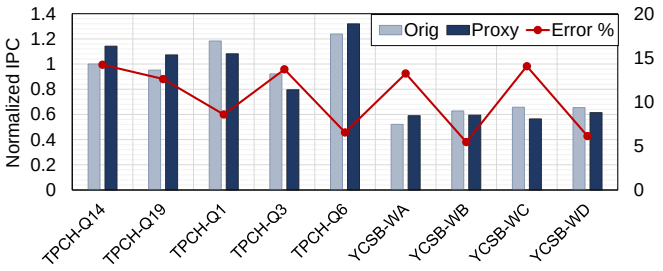
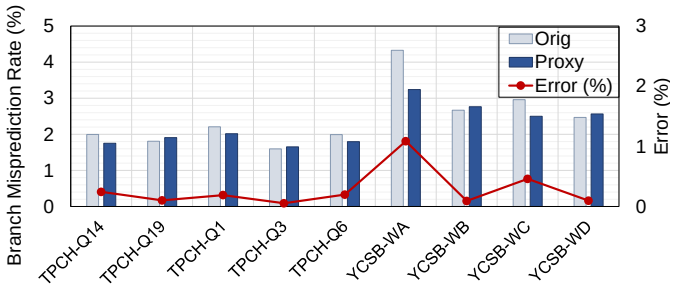


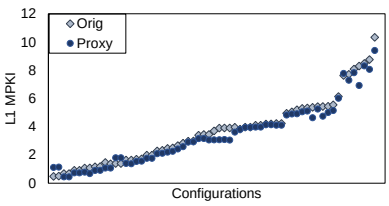
Fig. 4: DynInst format



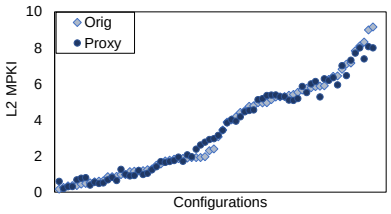
(a) Instructions per cycle: Varying core and pipeline configurations



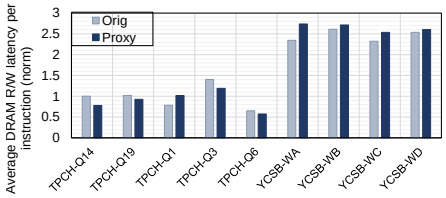
(b) Branch misprediction rate: Varying branch pred. configs



(c) L1 MPKI: L1 cache and prefetcher



(d) L2 MPKI: L2 cache and prefetcher



(e) DRAM R/W latency: Diff configs

Fig. 5: Evaluating core, branch predictor, cache, prefetcher and DRAM configurations using CAMP proxies

IV. Experiments and Results

For profiling and validation, we use a detailed architecture simulator Macsim [14], connected with DRAMSim2 [15] memory simulator. We evaluate CAMP using a set of big-data data-serving (Yahoo! Cloud Serving Benchmark (YCSB)[20]) and data-analytics applications (TPC-H benchmarks [21]). We run the standard benchmarks provided with YCSB framework, which cover the most important operations (read, write and insert) performed against a typical data-serving database. TPC-H models a decision-support system for order-processing engines, with queries performing different business-like analyses. We run 5 TPC-H benchmark queries. Both TPC-H and YCSB benchmarks interact with a backend MySQL database. The test databases are chosen to have a total size of $\sim$ 10-12GB so that the data fits into memory of the server nodes. We fast forwarded each benchmark to skip the initialization stage and clone one particular phase of the application consisting of 1 billion instructions (to capture other phases, we can choose other 1 billion instruction windows). It should be noted that profiling is a one-time cost and CAMP receives only a statistical profile as input (independent of the execution length). The system configuration used for collecting CAMP profiles is shown in Table II. We evaluate CAMP's accuracy in predicting various performance metrics across different core, pipeline, branch predictor, cache and memory configurations.

Core configurations - First, we compare the effectiveness of CAMP proxies in replicating overall performance of the database applications. We evaluate 8 different core configurations per benchmark by changing the pipeline width between 2-8, re-order buffer size between 128-512 and issue rate between 2-4. Figure 5a shows the results. We can observe that the average error between the proxy and original applications is $\sim$11%. Highest error is experienced by the TPCH-Q14 benchmark as it suffers from aliasing effects in the stride history table due to complex join-based access patterns, leading to higher L1/L2 cache and memory performance cloning errors. Increasing the stride history length can improve memory performance cloning accuracy but at the expense of higher metadata overhead. Overall, CAMP's methodology of capturing different instruction and memory access locality metrics leads to small error rates across most benchmarks (including complex queries in TPC-H and YCSB benchmarks). Overall, the proxies replicate the overall performance behavior with $\sim$0.94 correlation.

TABLE II: Profiled system configuration

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

Branch predictor configurations - Next, we evaluate the effectiveness of CAMP proxies in replicating behavior of original applications across different branch predictor configurations. In particular, we test two different branch predictors (gshare and tournament) and we also vary the predictor's branch history depth between 8-18. Figure 5b shows the average error in branch misprediction rate between the original and proxy applications. We can see that the average error is less than 1% (correlation = 0.93). This shows that CAMP's methodology of using branch transition rates to track predictability of control instructions is fairly accurate to model application behavior across different branch predictor configurations.

L1 cache and prefetcher configurations - Next, we evaluate CAMP's effectiveness across different L1 cache and prefetcher configurations. We evaluate 6 different L1 cache configurations per benchmark (varying the cache size from 16KB-64KB, associativity from 2-8). For each cache configuration, we also vary the L1 stream prefetcher configurations by changing the stream detection window between 8-16, prefetch degree between 0-4. Results showing the L1 cache miss rate errors is shown in Figure 5c. Capturing both temporal and spatial locality patterns using long history-based stride transitions in the memory access streams of the complex, big-data applications leads to highly accurate replication of cache performance. The proxies experience about an error up to 2 MPKI in some configurations, especially when cache line size of L1 caches changes significantly because the collected memory profiles do not capture locality within cache-blocks. Nonetheless, we can observe that the overall correlation between the proxy and original applications is high (0.98).

L2 cache and prefetcher configurations - Next, we evaluate CAMP's effectiveness across different L2 cache and prefetcher configurations. We evaluate 8 different L2 cache

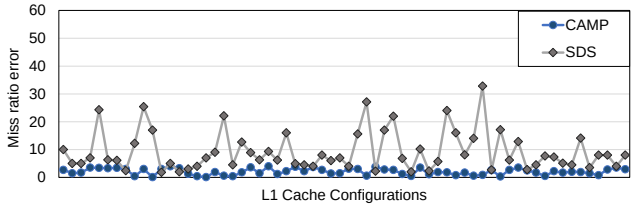


Fig. 6: Comparing L1 miss ratio cloning error between CAMP and SDS proxies for various L1 cache configurations.

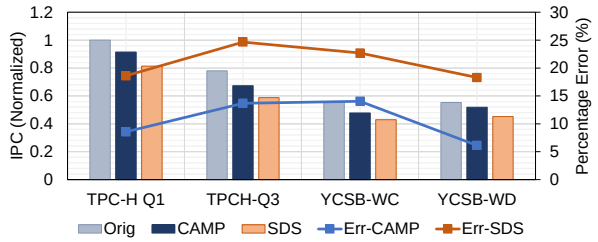


Fig. 7: IPC cloning accuracy of CAMP versus SDS proxies.

configurations per benchmark (varying the cache size from 128KB-1MB, associativity from 2-16). For each cache configuration, we vary the L2 stream prefetcher configurations by changing the stream detection window between 8-16, prefetch degree between 0-2. Results showing the L2 cache miss rate errors is shown in Figure 5d. We can observe that the overall correlation is high (0.97) due to accurate modeling of load-store patterns and write-back cache traffic.

DRAM performance - Next, we evaluate effectiveness of the CAMP proxies to enable design-space exploration of the memory system in lieu of the original applications. In particular, we evaluate 6 different DRAM configurations (changing the memory controller scheduling policy between FR-FCFS and FCFS, channel parallelism between 4-8, row buffer size between 2-4 KB) per benchmark (total 54 configurations). We compare the original and proxy benchmarks in terms of average read/write latency per instruction (see Figure 5e). Each value is normalized with the original TPC-H Q14 benchmark's performance metrics. Overall the average error in average read-write latency per instruction is 14.5% (correlation = 0.89).

Comparison with prior techniques - Figure 6 compares the clones generated using the single dominant stride (SDS proxy) profile, the most commonly used statistic in literature for modeling memory locality patterns in system-level proxy benchmarks, against CAMP proxies. For this, we vary the L1 cache size from 16-64KB and associativity between 2-8. We observe that the SDS clones show significant errors in L1 miss ratio at many points, reaching as high as 33%. CAMP proxies, on the other hand, show $\leq 1\%$ errors in most cases, and only a few data points have relatively higher error ($\leq 4\%$). This result demonstrates that the SDS approach is not suitable for modeling complex memory access patterns of big-data workloads. Figure 7 compares the IPC cloning accuracy of the SDS and CAMP proxies for three big-data benchmarks across different core pipeline and cache configurations. Overall, by accounting for accurate memory locality models together with replicating the program ILP, instruction types, basic block etc., CAMP proxies achieve much lower cloning error ($\sim 11\%$) compared to the SDS proxies ($\sim 21\%$).

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

V. Conclusion

In this paper, we proposed a methodology (CAMP) to solve the confidentiality and representativeness problems of workload performance cloning for big-data applications. CAMP accurately models both core-performance and memory locality, along with modeling the feedback loop between the two. To model the core performance, we adopt existing methods for generating proxy instruction streams by capturing and modeling the dependencies between instructions, instruction types, etc. We add an improved memory locality profiling approach that captures both the spatial and temporal locality of applications. Finally, we introduce a novel proxy generation and replay technique that integrates the core and memory locality models together to create accurate system-level proxy benchmarks. We demonstrate that CAMP clones can mimic the original application's performance behavior and that they can capture the performance feedback loop well. For a variety of real-world database applications, we show that CAMP achieves an average cloning error of $\sim$11%. We believe this is a new capability that can enable accurate overall system (core and memory subsystem) design exploration.

VI. Acknowledgement

The authors of this work are supported partially by SRC under Task ID 2504 and National Science Foundation (NSF) under grant number 1337393. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other sponsors.

References

[1] A. Joshi *et al.*, "Performance cloning: A technique for disseminating proprietary applications as benchmarks," in *IISWC*, 2006, pp. 105–115.
[2] K. Ganesan *et al.*, "Synthesizing memory-level parallelism aware miniature clones for spec cpu2006 and implantbench workloads," *ISPASS*, 2010.
[3] R. Panda and L. John, "Proxy benchmarks for emerging big-data workloads," in *IEEE PACT*, 2017.
[4] G. Balakrishnan and Y. Solihin, "West: Cloning data cache behavior using stochastic traces." *HPCA*, pp. 387–398, 2012.
[5] A. Awad and Y. Solihin, "Stm: Cloning the spatial and temporal memory access behavior." *HPCA*, pp. 237–247, 2014.
[6] M. Ferdman *et al.*, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," *SIGPLAN Not.*, vol. 47, no. 4, pp. 37–48, Mar. 2012.
[7] R. Panda, C. Erb, M. Lebeane, J. Ryoo, and L. K. John, "Performance characterization of modern databases on out-of-order cpus," in *IEEE SBAC-PAD*, 2015.
[8] R. Panda and L. K. John, "Data analytics workloads: Characterization and similarity analysis." in *IPCCC*. IEEE, 2014, pp. 1–9.
[9] R. H. Bell, Jr. and L. K. John, "Improved automatic testcase synthesis for performance model validation," in *ICS*, 2005, pp. 111–120.
[10] A. Joshi, L. Eeckhout, and L. John, "The return of synthetic benchmarks," in *Proceedings of the 2008 SPEC Benchmark Workshop*, 1 2008.
[11] "SPEC CPU 2006 Benchmarks," www.spec.org/cpu2006.
[12] Z. Jin and A. C. Cheng, "Implantbench: Characterizing and projecting representative benchmarks for emerging bioimplantable computing." *IEEE Micro*, 2008.
[13] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *SIGOPS Oper. Syst. Rev.*, 2002.
[14] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "Macsim: A cpu-gpu heterogeneous simulation," *User Guide, Georgia Inst. of Tech.*, 2012.
[15] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, pp. 16–19, Jan. 2011.
[16] M. Haungs, P. Sallee, and M. K. Farrens, "Branch transition rate: A new metric for improved branch classification analysis." in *HPCA*, 2000, pp. 241–250.
[17] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, Jun. 1970.
[18] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable & accurate parallel multi-core simulations," in *SC*, 2011.
[19] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.
[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010.
[21] "TPC-H Benchmark Suite," http://www.tpc.org/tpch.