Proxy Benchmarks for Emerging Big-data Workloads

Reena Panda
University of Texas at Austin
reena.panda@utexas.edu

Lizy Kurian John
University of Texas at Austin
ljohn@ece.utexas.edu

***Abstract*—Early design-space evaluation of computer-systems is usually performed using performance models such as detailed simulators, RTL-based models etc. Unfortunately, it is very challenging (often impossible) to run many emerging applications on detailed performance models owing to their complex application software-stacks, significantly long run times, system dependencies and the limited speed/potential of early performance models. To overcome these challenges in benchmarking complex, long-running database applications, we propose a fast and efficient proxy generation methodology, PerfProx that can generate miniature proxy benchmarks, which are representative of the performance of real-world database applications and yet, converge to results quickly and do not need any complex software-stack support. Past research on proxy generation utilizes detailed micro-architecture independent metrics derived from detailed functional simulators, which are often difficult to generate for many emerging applications. PerfProx enables fast and efficient proxy generation using performance metrics derived primarily from hardware performance counters. We evaluate the proposed proxy generation approach on three modern, real-world SQL and NoSQL databases, Cassandra, MongoDB and MySQL running both the data-serving and data-analytics class of applications on different hardware platforms and cache/TLB configurations. The proxy benchmarks mimic the performance (IPC) of the original database applications with ∼94.2% (avg) accuracy. We further demonstrate that the proxies mimic original application performance across several other key metrics, while significantly reducing the instruction counts.**

I. Introduction

Data handling and management has become an integral component of all businesses, big or small. Every major industrial sector, be it health-care, scientific-computing, retail, telecommunication, social networking etc., generates large amounts of data every day. For example, Facebook reported to have 21 peta-bytes of data in 2010, with 12 tera-bytes of new data added everyday and 800 tera-bytes of compressed data scanned daily [1]. Thus, recent times have seen an unprecedented boom in the need for efficient data management systems and applications. While traditional data management systems were based on the structured-query language (SQL) based relational databases, a new group of databases popularly known as NoSQL databases have recently emerged as competitive alternatives owing to their simplicity, flexibility and scalability properties [2], [3], [4], [5], [6], [7], [8]. Big-data processing needs not only challenge the capabilities of database management systems, but also pose significant challenges to the conventional computing systems to efficiently process data in terms of both performance and power. Thus, computer designers need to re-evaluate their design principles to target database applications.

Early computer design evaluation is performed using performance models such as execution-driven simulators or RTL-based models. However, several emerging applications are often complex targets to evaluate on early performance models owing to their complex application software-stacks, significantly long run times, system dependencies, etc. Figure 1 shows the software stack of a typical web-serving engine, consisting of layers of complicated software levels interacting together to form the backbone of the engine. Running similar applications requires handling different software layers, back-end databases, third-party libraries, etc., which is extremely challenging (often impossible) to support on early performance models. Furthermore, detailed performance models are significantly slower than real hardware that makes it difficult to analyze complete execution characteristics of such long-running applications.

Typically, a set of standard benchmarks are used for performing computer design-space exploration. Cloudsuite [9] and BigDataBench [10] are recently proposed benchmark suites consisting of a set of real-world and synthetic applications representing the broad space of emerging big-data applications. Few other research studies [11], [12], [13], [14] have explored to simplify database benchmarking by using smaller data-sets etc. However, these efforts suffer from similar challenges as the real-world applications, i.e., they rely on the ability of early performance models to support complex software stacks with back-end databases. On the other end of the spectrum are benchmarks like SPEC CPU2006 [15], MiBench [16], etc., which are comparatively simpler targets for performance evaluation but several recent research studies [10], [17], [18] have demonstrated that their performance behavior is very different from many big-data applications.

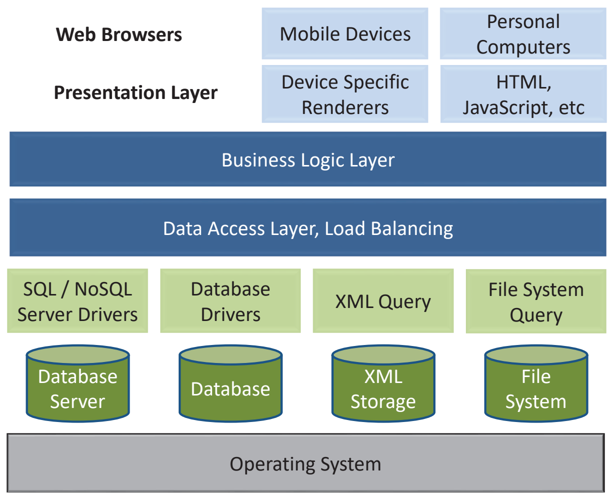


Fig. 1: Software stack of a typical web-serving engine

In order to overcome the difficulties in simulating emerging applications, in this paper, we propose a fast and efficient proxy synthesis methodology, **PerfProx**, to create miniature proxy benchmarks that are representative of the performance of real-world database applications and yet, converge to results quickly and do not need any complex software-stack support. The proposed methodology achieves a sense of representativeness by synthesizing proxies based on detailed workload characterization of the low-level dynamic runtime characteristics of the database applications. The workloads, which we considered in this paper include data-analytics and data-serving applications using SQL and NoSQL-based backend databases (e.g., Cassandra, MongoDB, etc.). Past research on proxy generation [19], [20] utilizes micro-architecture independent metrics derived from detailed functional simulators, which are often very difficult to generate for such emerging applications. Also, program profilers (e.g., Pin [21]) often face difficulties when dealing with Java and databases. PerfProx enables fast and efficient proxy generation for such applications using performance metrics derived primarily from hardware performance counters. Many big-data workloads do not work reliably with many profiling tools and thus, performance-counter based characterization and associated extrapolation into generic parameters that the code generator can take is an important contribution. For database applications that can work with fast program profilers (e.g., Pin), PerfProx augments its memory access modeling methodology by capturing detailed patterns in the original memory access streams. The key contributions made in this paper are as follows:

- We propose to generate proxy benchmarks to simplify benchmarking of big-data data-serving and data-analytics workloads without needing back-end database support.
- We present a proxy benchmark generation methodology, PerfProx, which enables fast and efficient proxy generation using performance metrics derived primarily from hardware performance counters.
- We evaluate the effectiveness of the proxy benchmarks using three real-world, popular databases, Cassandra [22], MongoDB [23] and MySQL [24] for both data-serving and data-analytics applications running across different hardware platforms and several cache/TLB configurations. We demonstrate that the proxy benchmarks closely follow the performance and power behavior of original applications while significantly reducing the instruction counts; the mean error in IPC between the proxies and the database applications is 5.1% for data-serving applications and 6.5% for data-analytics applications.
- Unlike prior proposals ([19], [20], [25], [26], [27], [28], [29]), we validate the performance of the proxies running on real hardware systems over their complete execution. To the best of our knowledge no prior proxy generation method has been cross-validated on real hardware. Also, unlike prior work which focused on Alpha/Sparc-based proxies, we generate x86-based proxy benchmarks which can be easily run on any x86-based simulator or real machines.

The rest of this paper is organized as follows: In section II, we will present a detailed description of the proposed proxy generation methodology. Section III presents an overview of the evaluated databases and benchmarks. Section IV provides details about our experimental methodology. We discuss key results and analysis in section V. Finally, we discuss related work in section VI, before concluding the paper in section VII.

II. Methodology

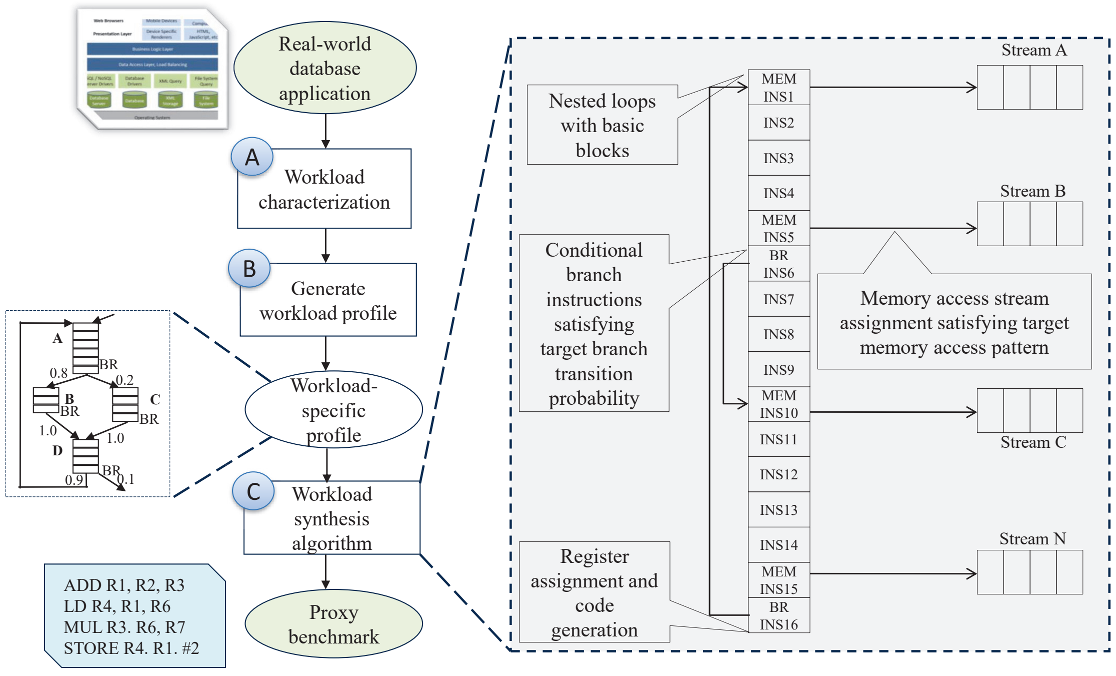
The main problem we want to address in this paper is that we want to generate miniaturized proxy benchmarks that have similar performance characteristics as the original real-world/complex workloads, yet are short-running and do not require any complex software-stack support. Furthermore, the proxy benchmarks should not divulge any proprietary information regarding the original end-user workload, which can motivate increased code sharing of real-world, end-user workloads with computer system designers and researchers, without any proprietariness or confidentiality concerns.

The overall framework of the proposed proxy benchmark generation approach is shown in Figure 2. PerfProx first characterizes the database applications running on real hardware and extracts their key performance metrics (step Ⓐ). During the workload characterization step, PerfProx captures low-level dynamic runtime characteristics of the program (like statistical simulation), adding accurate instruction-locality, memory access and branching models. Based on the extracted performance features, PerfProx builds a ***workload-specific profile*** for each database application that uniquely summarizes the application's runtime behavior over its entire execution time (step Ⓑ). Synthesizing using statistics rather than the original application source code effectively hides the functional meaning of the code/data, which addresses any proprietariness or confidentiality concerns about sharing end-user workloads. Finally, PerfProx's workload synthesizer uses the captured workload-specific profiles to generate the proxy benchmarks, which have similar features as original application (step Ⓒ).

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

A. *Workload Characterization*

As discussed before, PerfProx monitors the runtime behavior of an application and produces a set of workload characteristics representing its low-level dynamic execution characteristics. PerfProx captures the execution characteristics of database applications primarily using hardware performance counters running on real hardware systems. It then extrapolates the performance counter data using analytical models to derive features representing the workload-specific profile.


Real-world database application
A
Workload characterization
B
Generate workload profile
Workload-specific profile
C
Workload synthesis algorithm
Proxy benchmark
A
BR
0.8
0.2
B
BR
C
BR
1.0
1.0
D
BR
0.9
0.1
Nested loops with basic blocks
Conditional branch instructions satisfying target branch transition probability
Register assignment and code generation
MEM INS1
INS2
INS3
INS4
MEM INS5
BR INS6
INS7
INS8
INS9
MEM INS10
INS11
INS12
INS13
INS14
MEM INS15
BR INS16
Stream A
Stream B
Memory access stream assignment satisfying target memory access pattern
Stream C
Stream N

```
ADD R1, R2, R3
LD R4, R1, R6
MUL R3. R6, R7
STORE R4. R1. #2
```

Fig. 2: Proxy generation methodology and workload synthesis algorithm

The workload-specific profile serves as an input to the workload synthesis algorithm, which generates representative proxy benchmarks that closely resemble the performance of the original applications. Many emerging big-data workloads do not work reliably with many profiling tools and thus, performance-counter based characterization and associated extrapolation enables fast and accurate analysis and proxy generation for such applications. For database applications that can work with fast program profilers (e.g., Pin), PerfProx further augments its memory access modeling methodology by capturing micro-architecture independent patterns from the original memory access streams. The key performance features captured by the PerfProx's workload characterization model are described as follows. The abstract workload-specific profile generated based on the following features is shown in Table I.

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

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

3) ***Control Flow Behavior****:* Another important metric that affects application performance significantly is its control flow performance. Poor branch prediction rates owing to hard-to-predict branches, irregular control flow behavior etc, can cause significant performance degradation in applications due to increased number of wrong path executions, wasted resources, etc. Prior research [30], [19], [25] has shown that an application's branch misprediction rate is highly correlated with the transition rate (switching probability) of the component branch instructions [31]. PerfProx estimates the overall branch predictability of an application in a directly correlated fashion based on the application's branch misprediction rate (measured using hardware performance counters). To model a target branch predictability into the proxy benchmark during proxy generation, PerfProx estimates the fraction of control instructions in the proxy benchmark that will have a particular predictability behavior. For example, assuming a 2-bit saturating counter based predictor, 100% and 50% branch

TABLE I: Workload-specific model

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

predictability can be modeled using a branch instruction which is mostly not-taken and a branch instruction which alternates between the taken and not-taken paths respectively. Similarly, a very hard-to-predict branch can be modeled to switch between the taken and not-taken paths in a random fashion.

*4) **Memory-access Model**:* Several prior research studies have shown that memory performance has a significant impact on database application's overall performance [32], [9], [18]. Although PerfProx's primary objective is to develop a fast and light-weight methodology to model application performance, it is crucial for PerfProx to model the cache and memory performance accurately. The principle of data locality and its impact on cache and memory performance is widely recognized. PerfProx models the data memory accesses using simple, strided stream-classes over fixed-size data arrays. PerfProx leverages a methodology to infer the memory stream strides based on the data cache miss rates of the original application (similar approach as [19], [33]). It employs a pre-computed table that holds the correlation between L1/L2 cache hit rates and the corresponding stride values. Particular memory access strides are determined, using the target L1/L2 hit rate information along with this table, by first matching the L1 hit rate of the memory operation, followed by the L2 hit rate. For example, memory accesses with 100% and 50% hit-rates can be modeled using a stride of zero and 8 respectively (assuming 64B cacheline size). Stride values are optimized to achieve the greater correlation of proxies in terms of target cache performance. Although approximate, such a mechanism to model strides based on the cache miss rates enables fast and efficient memory pattern modeling of complex workloads, which are otherwise difficult to simulate on detailed performance simulators. This technique was used to estimate memory access strides for database applications (e.g., JAVA-based Cassandra) that often can not reliably run to completion using program profilers.

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

Furthermore, it has been shown by several prior research studies that database applications tend to have higher TLB misses (often as frequent as cache misses) [18], [9], which has a significant impact on their performance. As discussed before, PerfProx models the data memory accesses using simple strided stream classes over fixed-size data arrays. In order to model the effects of TLB performance, PerfProx controls the degree of concurrency in its active memory access streams, i.e, it controls the number of unique memory streams actively accessed by the proxy application within a fixed window of instructions. Individual load/store instructions are assigned to different active data streams based on this concurrency factor. The proxy data footprint is also scaled according to the target data-set size of the database application.

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

*6) **System Activity**:* Prior research [18], [9] has shown that the emerging database applications spend a significant fraction of their execution time executing operating system (OS) code, which has a significant impact on their overall performance. To model the performance impact of high system activity, PerfProx measures the system activity in the original applications using *STRACE* tool and the fraction of executed user-mode and kernel instructions using hardware performance counters. During proxy generation, PerfProx inserts corresponding desired fraction of system calls into the basic blocks in the proxy benchmark to achieve the desired level of system activity.

B. Synthetic Proxy Benchmark Generation

In this section, we will discuss the workload synthesis algorithm (see Figure 2). The workload synthesis algorithm takes as an input the workload-specific profile captured during the workload characterization phase (see Table I for a list of the captured metrics). The proxy benchmark generation steps are listed in Algorithm 1. PerfProx first estimates the total number of static basic blocks to instantiate in the proxy benchmark. It then chooses a random number in the interval [0, 1] to select a basic block based on its access frequency. The size of the basic block (in terms of number of instructions) is chosen to satisfy the mean and standard deviation of the target basic block size (line 6). The IMIX statistics are used to populate the basic block with appropriate instruction types (line 8), while ensuring that the last instruction of every basic block is a conditional branch instruction (line 13). Every instruction is assigned a dependency distance (i.e., a previous instruction that generates its data operand) to satisfy the dependency distance criterion. The memory instructions are assigned a stride and memory access stream based on the memory model described before. System calls are injected (or not) into the basic block based on the target system-call frequency. Finally, an X86 test operation is inserted before the branch instruction to set the condition codes that affect the conditional branch outcome. The test instruction's operand is chosen to control the branch transition rate in order to satisfy the target transition rate of every basic block. These steps are repeated till we generate the target number of static basic blocks. Finally, architectural register operands are assigned to each instruction to satisfy the dependencies in step 9 (line 16).

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

Algorithm 1 Workload synthesis algorithm

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

volatile directive for each *asm* statement prevents the compiler from optimizing out the program machine instructions.

C. *Discussion*

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

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

III. Databases and Benchmarks

In this section, we will discuss the databases and benchmarks used for evaluating the proposed PerfProx methodology.

A. *Databases*

In this paper, we evaluate three NoSQL and SQL databases (Cassandra, MongoDB and MySQL), which are discussed in the following paragraphs.

*1) **Cassandra**:* Apache Cassandra [22] is a popular, Java-based column-family style NoSQL database. It is incrementally scalable, eventually consistent, and has no single point of failure. Every node in the Cassandra cluster knows of and has the key for at least one other node and any node can service a request. The node structure can be visualized as a ring/web of interconnected nodes. Cassandra is semi-structured i.e., its data may share some of the same fields, or columns, but not all. In this way Cassandra is slightly more organized than MongoDB, but still not as rigid as MySQL.

*2) **MongoDB**:* MongoDB [23] is an open-source, C++ based document-style NoSQL database. It is designed for speed and scalability. It has a flexible schema (allows objects to not have fixed schema/type) and can store large documents such as binaries, images etc. Documents are stored as binary JSON objects and may be organized into collections. Within a collection each document has a primary key, and an index can be created for each query-able field. MongoDB's data is searched using keys and meta-data information.

*3) **MySQL**:* MySQL [24] is one of the worlds most popular open-source relational database management system. It enables the cost-effective delivery of reliable, high-performance and scalable web-based and embedded database applications. MySQL is designed to work on data whose fields are predefined and finite in number. Given this regular layout, MySQL can organize and search through data in multiple dimensions. This is both its strength and limitation, as it can't use the same strategy on less structured data.

B. *Benchmark Description*

Next, we will discuss the evaluated data-serving and data-analytics benchmarks in detail.

*1) **YCSB Benchmarks**:* We use the Yahoo! Cloud Serving Benchmark (YCSB)[4] to represent the data-serving applications using Cassandra, MongoDB and MySQL databases.

TABLE II: YCSB core workloads

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

TABLE III: TPC-H benchmark description

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

TABLE IV: System description

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

YCSB is a standard benchmarking framework that is used to evaluate different cloud systems. YCSB's framework consists of a workload generating client and a set of standard 'core' workloads (see Table II), which cover the most important operations performed against a typical data-serving database. Our test database is generated using the YCSB framework and has over 10 million records (total size is $\geq$ 12GB). The data-set size is chosen so that the data fits into memory of the server nodes, which is the recommended operational setup for scale-out applications for better performance [34]. Every test run performs 1 million operations against the database.

*2) **TPC-H Benchmarks**:* We use TPC-H benchmarks [35] to represent the data-analytics applications. TPC-H models a decision-support system environment for commercial order processing engines. It consists of a set of queries that interact with the server system to perform different business-like analyses. Similar to Barroso et al. [12], we use a data-set size ($\sim$10GB) to analyze the behavior of an in-memory database. We used dbgen and qgen tools (provided on TPC's website) to create/populate the database and generate the queries. We run 5 queries from the TPC-H benchmark suite on MySQL database. Query details are shown in Table III.

IV. Experimental Methodology

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

We use MongoDB version 2.6.5, running one mongod instance per server node. MongoDB's config server and router node are setup on the server node, we also verified that the router node and config server processes were light-weight and were not bottlenecks in our tests. We use Cassandra version 0.1.7 with Java Oracle JDK version 1.7 and a JVM heap size of 8GB. We use MySQL version 5.1.15.

The proxy benchmarks are compiled using gcc with the -O0 optimization flag to avoid compiler optimizations that remove dead-code or other optimizations that alter the inserted code. In order to evaluate microarchitectural performance of the actual applications and corresponding proxy benchmarks, we use Linux perf tool [36] that provides an interface to the processor performance counters. We also use Intel's PIN tool [21] for workload characterization.

V. Results and Analysis

In this section, we evaluate the effectiveness of the proxy benchmarks in mimicking the behavior of the original database applications based on several key performance metrics across different systems. In the following sections, YCSB benchmarks are represented as DB-WLx, where DB is the original database name and x is the YCSB workload (A-D). Also, database and proxy benchmark results are represented as Actual (A) and Proxy (P) respectively. Apart from comparing the error between different performance metrics of the proxy and database applications, we also compare the Pearson's correlation coefficient (ρ) for each performance metric. Pearson's correlation coefficient indicates how well the proxy benchmarks track the trends in the actual database applications, with 1 indicating a perfect correlation, and 0 indicating no correlation.

*A. **Proxy Performance Validation on System-A***

Figure 3 compares the instructions per cycle (IPC) of Cassandra, MySQL and MongoDB databases running the YCSB and TPC-H benchmarks along with their corresponding proxies on system-A. We can see that IPC of the proxy benchmarks closely follow the IPC of the original applications, with a high correlation (ρ = 0.99). The mean error between the proxy IPC and actual application IPC is 6.1% approximately (10.7% max) across all workloads. Considering the data-serving applications only, the average error in IPC between the proxy and the actual applications is 5.1%. MongoDB experiences worse errors as compared to Cassandra and MySQL. Performance of MongoDB-based applications are impacted by their cache and TLB performance [18]. Because of PerfProx's simple memory access locality modeling technique, PerfProx proxies experience higher deviation in terms of the cache and TLB performance with respect to the original applications, which results in the higher overall performance modeling error. The data-analytics applications have an average error of 6.5% between the proxy and actual applications. Next, we will analyze

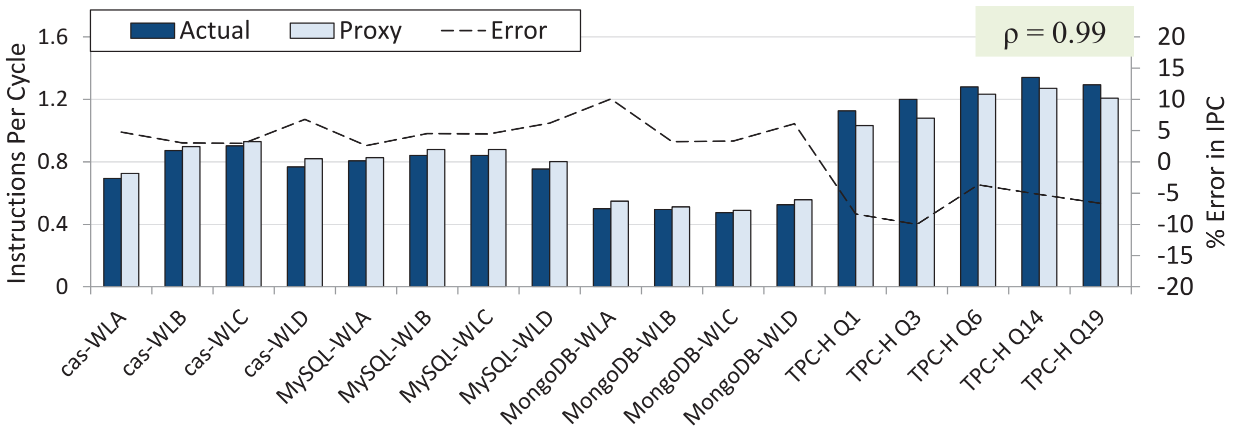


Fig. 3: IPC of real databases and proxy applications

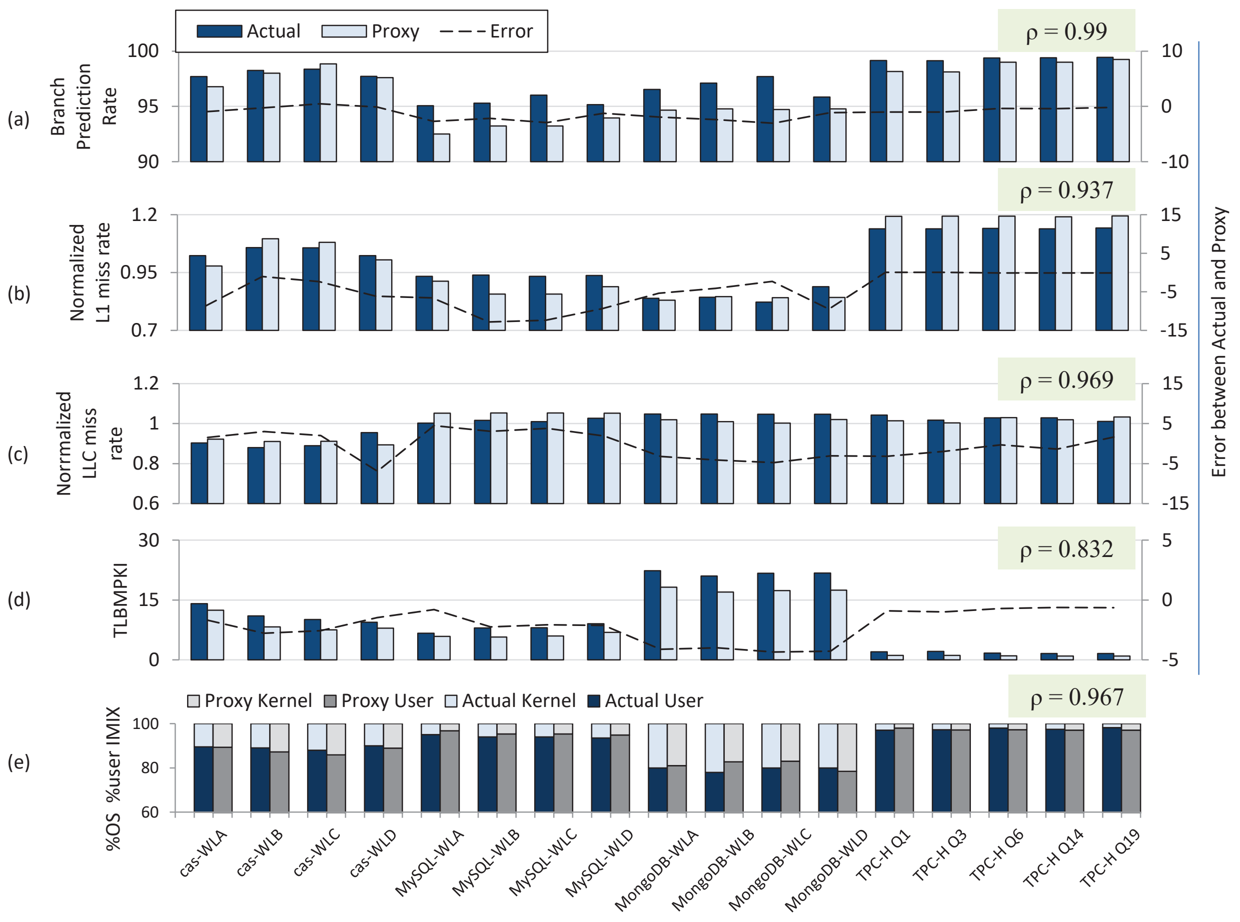


Fig. 4: Comparison of performance features of original and proxy applications on system-A. Error % on the right side axis.

performance of several key performance metrics, which lead to the overall performance correlation between the original and proxy benchmarks.

Figure 4a compares the branch prediction rates of the original and their corresponding proxy benchmarks. We can observe that the error between the branch prediction rates of the proxy benchmarks and the actual applications is small (average error = 1.5%, ρ = 0.99). This shows that PerfProx's methodology of capturing and mimicking branch transition rates is effective at achieving the target branch prediction rates fairly accurately. Figures 4b and 4c compare the L1 cache and last-level cache (LLC) hit rates respectively for Cassandra, MySQL and MongoDB databases running the YCSB and TPC-H benchmarks with their corresponding proxies (normalized with respect to the cache hit rate of Cassandra running YCSB WLA benchmark). The average error in mimicking L1 and LLC Cache hit rate is 6.1% and 3.1% respectively. In terms of TLB behavior (Figure 4d), the average error between the proxy and original applications is higher as compared to other performance metrics. Nonetheless, the trend in TLB performance is captured to a reasonable degree across the different workloads (ρ = 0.83). Similarly, in terms of system activity (Figure 4e), the fraction of user to system instructions in the proxy benchmarks closely follows the original applications, with a correlation (ρ) of 0.967.

B. *Proxy Cross-platform Validation on System-B*

In this section, we evaluate the performance sensitivity of proxy benchmarks generated from system-A on the system-B micro-architecture (see Table IV).

Figure 5a shows the IPC of the proxy versus actual applications on system-B for Cassandra-based applications (normalized with respect to actual Cas-WLA). We can observe that IPC of proxy benchmarks experience an average error of ~19.4% in replicating original application performance across the different YCSB workloads. As PerfProx's workload features are derived using microarchitecture-dependent characterization (e.g., cache miss rates etc) based on a target system, the performance correlation of the proxies on similar machines is higher. However when tested on machines with very different configurations, the performance correlation of proxies comparatively degrades. We also compare the original and the proxy workloads using several other key metrics e.g., L2 misses per kilo instructions (MPKI), LLC MPKI and branch prediction rate (see Figures 5b, 5c and 5d). We can observe that although the L2 and LLC MPKI of the proxy benchmarks follow the performance trends of the original applications, the degree of correlation is lower because of dependence of profiled cache performance metrics on the profiled cache hierarchy.

In Figure 5e, we compare the IPC of the proxy benchmarks

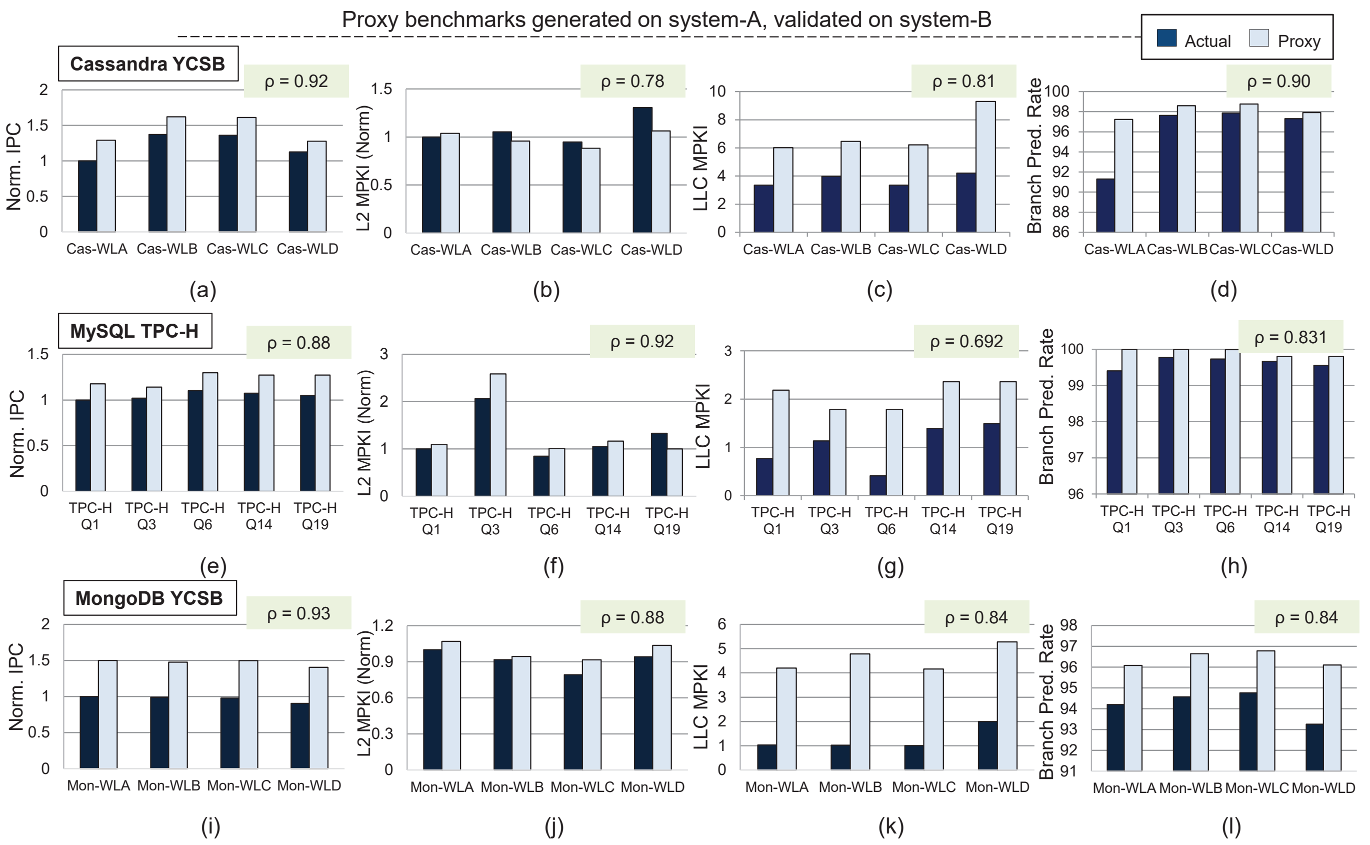


Fig. 5: Proxies from system-A validated on system-B: (a) IPC, (b) L2 MPKI, (c) LLC MPKI, (d) Branch prediction rate

versus the original TPC-H applications on system-B. We also compare the proxy and original workloads across several other key metrics, L2 MPKI (average error = ∼0.78 MPKI), LLC MPKI (average error = ∼1 MPKI) and branch prediction rate (average error = 0.25%) as shown in Figures 5f, 5g and 5h respectively. The L1 cache and TLB performance (not shown here due to space considerations) also have similar correlations between the original and proxy applications. Memory locality modeling using stride-based patterns leads to accurate capture of application spatial locality, thereby improving the cache performance correlation between the proxy and actual applications. However, the assumed stride model fails to capture long-distance reuse locality of accesses, which manifests as slightly worse errors in modeling LLC locality. On the other hand, the cache miss rate based memory locality modeling technique captures reuse probability at lower level caches as well, leading to slightly better performance correlation. Future work will focus on incorporating longer-distance reuse locality patterns into the memory access model.

Finally, Figure 5i shows the IPC of the proxy benchmarks versus the original applications on system-B for MongoDB-based applications. Although the average error between the IPC of proxy benchmarks with respect to the original database queries on system-B is high, the proxy benchmarks still capture the IPC trends of the original application pretty well (average correlation = ∼0.93). We also compare the proxy and original workloads across several other key metrics, L2 MPKI, LLC MPKI (average error = ∼3 MPKI) and branch prediction rate (average error = 2.3%) as shown in Figures 5j, 5k and 5l respectively.

C. *Proxy performance sensitivity analysis on different cache/TLB configurations*

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

We first evaluate the performance sensitivity of the data-analytics applications. We choose to evaluate TPC-H Q19 as Q19's proxy experienced the highest error in replicating cache performance among the 5 TPC-H queries on system-A. We use a PIN-based cache simulator to measure the cache performance of the proxy and the original TPC-H queries across 20 different cache configurations, where we we vary the cache size between

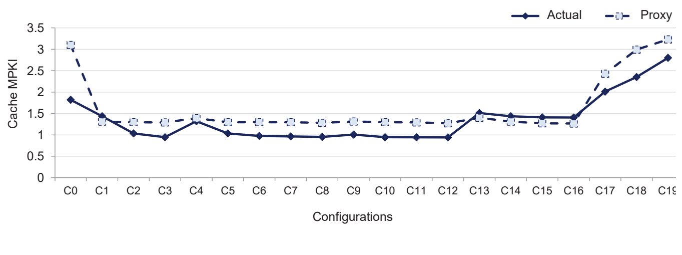


Fig. 6: Performance sensitivity of data-analytics (TPC-H Q19) proxy to different cache configurations

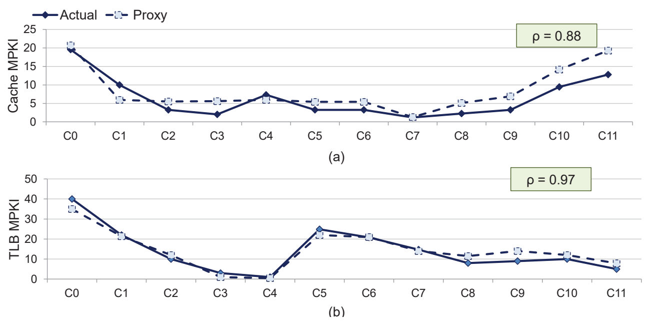


Fig. 7: Sensitivity of data-serving proxy performance to different cache and TLB configurations

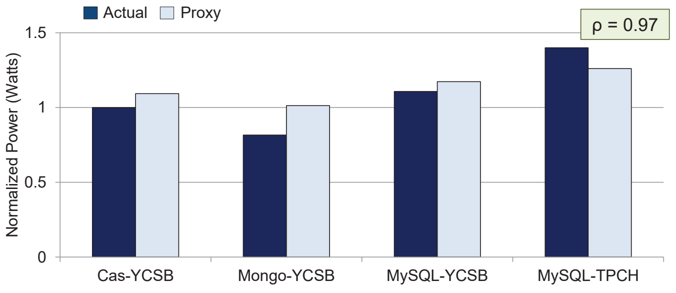


Fig. 8: Comparing power consumption of proxy versus actual applications

16-256KB and associativity between 1, … 32. Figure 6 shows the cache MPKI of the original TPC-H Q19 and its proxy for the different configurations. We can see that the cache MPKI of the proxy benchmark follows the original application with an average deviation of 0.5 MPKI and a high correlation of 0.89 across the different configurations.

We also evaluate the cache performance sensitivity of data-serving applications using the YCSB workload with MongoDB database. For the different cache configurations, we vary cache sizes between 16 to 256KB and associativity between direct-mapped, 2, … 16. Similarly, the different TLB configurations correspond to different TLB sizes (32 - 256 entries) and associativity (2 - 8). Figure 7 shows the cache and TLB MPKI sensitivity of the proxy and database application. We can observe that the cache MPKI of the proxy benchmark follows the original application closely across different cache configurations, with an average deviation of ∼2 MPKI. Similarly, the TLB MPKI of the proxy benchmarks follows the original application with a mean error of 2.2 MPKI. The proxies and the actual applications have a correlation of 0.88 and 0.97 with respect to cache and TLB performance respectively.

D. *Energy-efficiency Analysis*

Figure 8 shows the average power consumption (in watts) of the individual databases running the data-serving and data-analytics applications and their corresponding proxy benchmarks (normalized with respect to the actual YCSB benchmarks running on Cassandra). We measure power using the Intel's RAPL counters on system-B. We can see that the correlation between the average power consumption of the proxy and actual applications is significantly high (ρ = 0.97). The power consumption of an application is often highly correlated with its performance behavior [37]. Since the proxy benchmarks mimic the performance behavior of the original applications closely (in terms of IMIX, instruction dependencies, cache/memory behavior etc.), they closely mimic the power characteristics of the actual applications as well.

E. *Comparison with standard benchmarking suites*

In this section, we compare the performance correlation between the original database applications and three standard benchmarks, SPEC CPU2006 [15], SPECjbb2013 [38] and Linpack [39]. The kiviat plots shown in Figure 9a show the performance trends of the original database applications and their corresponding proxy benchmarks, while the kiviat plots in Figure 9b shows performance metrics corresponding to the standard benchmarks across several key metrics. Specifically, the kiviat plots are based on selected raw performance metrics (L1D, L1I, LLC, I/D TLB MPKI, %kernel instructions executed (Ker), branch misprediction rate (BrMis)), normalized by their maximum observed values on system-A. Modern database applications suffer from several bottlenecks which limit their overall performance on contemporary hardware systems. The plots illustrate significant diversity in the performance and bottlenecks of different database applications and standard benchmarks. For example, SPECjbb stresses a different set of system components (branch misprediction rate and LLC cache misses) than MongoDB applications. Even with a comparable dataset size (over 10GB), Linpack does not encounter similar memory subsystem issues as the database applications, demonstrating that Linpack program behavior is different from databases even when the data-set is big. The plots also show how closely the generated proxy benchmarks resemble performance trends of the original workloads. Thus, the proxy benchmarks can be used for effective performance validation, while being very simple targets for performance evaluation. Improving the memory and instruction locality models can further improve their fidelity.

F. *Degree of Miniaturization*

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

VI. Related Work

Several research studies [11], [12], [13], [14] use a set of simpler database queries or smaller data-sets to simplify benchmarking databases on early performance models. Dbmbench [13] scales down the TPC-H and TPC-C benchmarks based on the database size, workload complexity and concurrency level and produces μTPC-H and μTPC-C workloads respectively. However, these approaches suffer from the challenge of supporting complex software stacks of modern-day databases on early performance models. In contrast, PerfProx captures key performance features of real-world database applications and distills them into smaller proxies that can be evaluated without any back-end database support. MinneSPEC [40] is

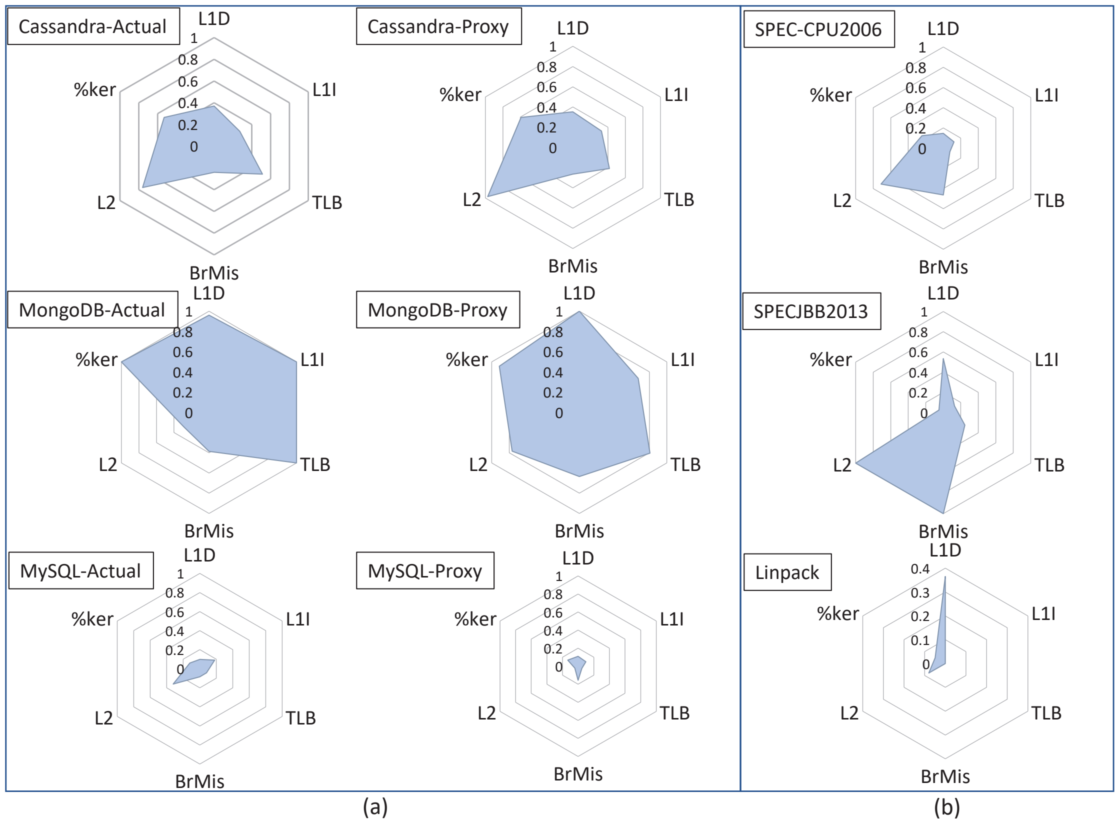


Fig. 9: Kiviat diagrams comparing performance of the original database applications, their proxy benchmarks and a set of standard benchmarks

another proposal to reduce simulation time of SPEC workloads by using miniature, representative input sets.

Bell et al. [19] proposed the black-box benchmark cloning approach using several execution-metrics. Other studies [20], [25] cloned proprietary applications into proxy benchmarks using only micro-architecture independent attributes. PerfProx enables fast and accurate database proxy generation by using hardware performance counters. Also, prior proposals were evaluated (often shorter run-times) for desktop-like benchmarks (SPEC CPU2000 [41], SPEC CPU2006 [15] etc) or embedded benchmarks (Implantbench [42]) on micro-architectural simulators. In contrast, PerfProx allows to monitor and analyze complete execution of database applications (with complex software stacks) on real systems. We also evaluate and validate the proxy performance on real hardware systems unlike prior proposals.

Simpoints [43] reduce simulation time by identifying regions of program execution with distinct performance characteristics. Using simpoint-based techniques for database applications requires supporting complete software stack of database applications on simulation frameworks and fast-forwarding support.

Trace-based schemes [44], [45], [46] have been explored to reproduce data-memory behavior of SPEC benchmarks specifically. As they don't model other workload features (instruction-side, control-flow etc), they are orthogonal to the proposed scheme. Also, the proposed approaches do not require storing large traces and are likely more portable (e.g., 32-bit to 64-bit).

VII. Conclusion

In this paper, we presented a novel methodology, PerfProx to generate representative proxy benchmarks that enable fast and efficient performance evaluation of emerging workloads without needing back-end database or complex software stack support. PerfProx generates proxies by monitoring and extrapolating database performance primarily using hardware performance counters. We evaluated the proxy benchmarking methodology using three popular and modern databases, Cassandra, MySQL and MongoDB for data-serving and data-analytics applications running across different hardware platforms and multiple cache/TLB configurations. We demonstrated that the proxy benchmarks closely follow the performance trends of database applications across several key performance metrics while significantly reducing the instruction count. The proxy benchmarks mimic the performance (IPC) of the original applications with 94.9% (average) accuracy for data-serving applications and 93.5% (average) accuracy for data-analytics applications.

VIII. Acknowledgement

The authors of this work are supported partially by SRC under Task ID 2504 and National Science Foundation (NSF) under grant number 1337393. We wish to acknowledge the computing time we received on the Texas Advanced Computing Center (TACC) system. 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. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10, 2010, pp. 1013–1020.

[2] M. Indrawan-Santiago, "Database research: Are we at a crossroad? reflection on nosql," in *NBIS*. IEEE Computer Society, 2012, pp. 45–51.

[3] A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang, "Can the elephants handle the nosql onslaught?" *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1712–1723, Aug. 2012.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010, pp. 143–154.

[5] A. Boicea, F. Radulescu, and L. I. Agapin, "Mongodb vs oracle – database comparison," in *EIDWT*. IEEE Computer Society, 2012, pp. 330–335.

[6] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *SIGMOD*. ACM, 2009, pp. 165–178.

[7] S. M. Y. Li, "A performance comparison of sql and nosql databases," in *PACRIM*, August 2013.

[8] D. Bartholomew, "Sql vs nosql," *Linux Journal*, p. 195, July 2010.

[9] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *ASPLOS*. New York, NY, USA: ACM, 2012, pp. 37–48.

[10] W. Gao, Y. Zhu, Z. Jia, C. Luo, L. Wang, Z. Li, J. Zhan, Y. Qi, Y. He, S. Gong, X. Li, S. Zhang, and B. Qiu, "Bigdatabench: a big data benchmark suite from web search engines," *CoRR*, vol. abs/1307.0320, 2013.

[11] K. Keeton and D. A. Patterson, "Towards a simplified database workload for computer architecture evaluations," in *In Workload Characterization for Computer System Design, edited byh*. Kluwer Academic Publishers, 2000, pp. 115–124.

[12] L. A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory system characterization of commercial workloads," *SIGARCH Comput. Archit. News*, vol. 26, no. 3, Apr. 1998.

[13] M. Shao, A. Ailamaki, and B. Falsafi, "Dbmbench: Fast and accurate database workload representation on modern microarchitecture," in *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '05. IBM Press, 2005, pp. 254–267. [Online]. Available: http://dl.acm.org/citation.cfm?id=1105634.1105653

[14] L. Van Ertvelde and L. Eeckhout, "Benchmark synthesis for architecture and compiler exploration," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–11.

[15] "SPEC CPU2006," https://www.spec.org/cpu2006.

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

[17] R. Panda and L. K. John, "Data analytics workloads: Characterization and similarity analysis." in *IPCCC*. IEEE, 2014, pp. 1–9.

[18] 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.

[19] R. H. Bell, Jr. and L. K. John, "Improved automatic testcase synthesis for performance model validation," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ser. ICS '05, 2005, pp. 111–120.

[20] A. Joshi, L. Eeckhout, R. H. B. Jr., and L. K. John, "Performance cloning: A technique for disseminating proprietary applications as benchmarks." in *IISWC*, 2006.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005, pp. 190–200.

[22] "Cassandra," wiki.apache.org/cassandra/FrontPage.

[23] "MongoDB," mongodb.org.

[24] "MySQL," http://www.mysql.com.

[25] K. Ganesan, J. Jo, and L. K. John, "Synthesizing memory-level parallelism aware miniature clones for spec cpu2006 and implantbench workloads." in *ISPASS*, 2010.

[26] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. Devito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *IPDPS*, ser. IPDPS '13, 2013, pp. 919–932.

[27] L. Eeckhout, R. H. Bell Jr., B. Stougie, K. De Bosschere, and L. K. John, "Control flow modeling in statistical simulation for accurate and efficient processor design studies," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA '04, 2004.

[28] M. Oskin, F. T. Chong, and M. Farrens, "Hls: Combining statistical and symbolic simulation to guide microprocessor designs," in *ISCA*, ser. ISCA '00, 2000, pp. 71–82.

[29] L. Eeckhout, K. de Bosschere, and H. Neefs, "Performance analysis through synthetic trace generation," in *ISPASS*, 2000, pp. 1–6.

[30] A. Joshi, L. Eeckhout, and L. John, "The return of synthetic benchmarks," in *Proceedings of the 2008 SPEC Benchmark Workshop*, San Francisco, CA, USA, 1 2008, pp. 1–11 (digitaal).

[31] M. Haungs, P. Sallee, and M. K. Farrens, "Branch transition rate: A new metric for improved branch classification analysis." in *HPCA*. IEEE Computer Society, 2000, pp. 241–250.

[32] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "Dbmss on a modern processor: Where does time go?" in *VLDB*, 1999, pp. 266–277.

[33] L. V. Ertvelde and L. Eeckhout, "Benchmark synthesis for architecture and compiler exploration," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, Dec 2010, pp. 1–11.

[34] "Mongodb architecture guide," http://www.mongodb.com.

[35] "TPC-H Benchmark Suite," http://www.tpc.org/tpch.

[36] "Linux perf tool," https://perf.wiki.kernel.org/index.php/Main_Page.

[37] K. Ganesan, J. Jo, W. L. Bircher, D. Kaseridis, Z. Yu, and L. K. John, "System-level max power (sympo): A systematic approach for escalating system-level power consumption using synthetic benchmarks," in *PACT*, 2010, pp. 19–28.

[38] "SPECjbb 2005," https://www.spec.org/jbb2005/.

[39] "HP Linpack," http://icl.eecs.utk.edu/hpl/.

[40] A. J. KleinOsowski and D. J. Lilja, "Minnespec: A new spec benchmark workload for simulation-based computer architecture research." *Computer Architecture Letters*, vol. 1, 2002.

[41] "SPEC CPU2000," https://www.spec.org/cpu2000.

[42] Z. Jin and A. C. Cheng, "Implantbench: Characterizing and projecting representative benchmarks for emerging bioimplantable computing." *IEEE Micro*, vol. 28, 2008.

[43] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. 5, pp. 45–57, Oct. 2002.

[44] G. Balakrishnan and Y. Solihin, "West: Cloning data cache behavior using stochastic traces." in *HPCA*. IEEE Computer Society, 2012, pp. 387–398.

[45] A. Awad and Y. Solihin, "Stm: Cloning the spatial and temporal memory access behavior." in *HPCA*, 2014, pp. 237–247.

[46] R. Panda, X. Zheng, and L. John, "Accurate address streams for llc and beyond (slab): A methodology to enable system exploration," in *IEEE ISPASS*, 2017.