Ditto: End-to-End Application Cloning for Networked Cloud Services

Mingyu Liang[*]
ml2585@cornell.edu
Cornell University
Ithaca, New York, USA

Yu Gan[*]
yg397@cornell.edu
Cornell University
Ithaca, New York, USA

Yueying Li
yl3469@cornell.edu
Cornell University
Ithaca, New York, USA

Carlos Torres
cltorres@meta.com
Meta
Menlo Park, California, USA

Abhishek Dhanotia
abhishekd@meta.com
Meta
Menlo Park, California, USA

Mahesh Ketkar
mahesh.c.ketkar@intel.com
Intel
Folsom, California, USA

Christina Delimitrou
delimitrou@csail.mit.edu
MIT
Cambridge, Massachusetts, USA

ABSTRACT

The lack of representative, publicly-available cloud services has been a recurring problem in the architecture and systems communities. While open-source benchmarks exist, they do not capture the full complexity of cloud services. Application cloning is a promising way to address this, however, prior work is limited to CPU-/cache-centric, single-node services, operating at user level.

We present Ditto, an automated framework for cloning end-to-end cloud applications, both monolithic and microservices, which captures I/O and network activity, as well as kernel operations, in addition to application logic. Ditto takes a hierarchical approach to application cloning, starting with capturing the dependency graph across distributed services, to recreating each tier's control/data flow, and finally generating system calls and assembly that mimics the individual applications. Ditto does not reveal the logic of the original application, facilitating publicly sharing clones of production services with hardware vendors, cloud providers, and the research community.

We show that across a diverse set of single- and multi-tier applications, Ditto accurately captures their CPU and memory characteristics as well as their high-level performance metrics, is portable across platforms, and facilitates a wide range of system studies.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; *n-tier architectures*; • **Computing methodologies → Modeling methodologies**; • **Software and its engineering →** *Software reverse engineering*; Software performance.

KEYWORDS

cloud computing, architecture, benchmarking and emulation, microservices, software reverse engineering

ACM Reference Format:
Mingyu Liang, Yu Gan, Yueying Li, Carlos Torres, Abhishek Dhanotia, Mahesh Ketkar, and Christina Delimitrou. 2023. Ditto: End-to-End Application Cloning for Networked Cloud Services. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3575693.3575751

[*]Equal contribution.

1 INTRODUCTION

Cloud computing now hosts a large fraction of the world's computation, ranging from machine learning workloads to latency-critical interactive services [23, 24]. Studying these applications is imperative to correctly design the systems that populate future cloud infrastructures.

There are three approaches to performing studies that require cloud applications; using real services (production or open-source) [43, 48, 53, 91, 98, 110], using simulation or trace replay [20, 22, 58, 82, 86, 109], and generating synthetic services that resemble the original in behavior and characteristics [18, 27, 56, 72, 87, 90]. All three approaches are subject to pitfalls.

Using real production services is, naturally, the most representative approach. Unfortunately production services are rarely publicly available, and open-source applications, although useful, often lack the complexity and update cadence of a real cloud deployment. Studies that rely on simulation or replaying traces from a production system offer some representativeness, but are tied to the system configuration the trace was collected on, and cannot easily generalize to arbitrary studies. Finally, generating synthetic benchmarks offers a middle ground, with the synthetic application capturing critical features of the original service, but being malleable enough to adjust to different studies. Unfortunately, most prior work on synthetic benchmark cloning is limited to CPU-centric, single-tier, and user-level applications [72, 87, 90].

Only capturing CPU-centric microarchitectural events is not enough to reproduce the performance and resource characteristics of cloud applications, which spend a large fraction of their execution in the networking stack and OS. Moreover, prior work on synthetic application cloning mostly considers generating assembly code to mimic metrics like IPC, cache miss rate, and dependency distance, but overlooks critical higher-level performance metrics, such as average and tail latency.

We present *Ditto*, an automated application cloning framework for end-to-end cloud services, designed for both monolithic applications and microservices. Ditto is the first system to clone an application's behavior across the system stack, including the hardware, I/O, networking layers, and OS. This is critical for cloud applications which spend a large fraction of their time at kernel level and the I/O stack [74]. It additionally also targets multi-tier microservices which span distributed deployments and are gaining in popularity.

Ditto relies on the following key techniques. First, it captures the dependency graph across distributed services using distributed tracing [4, 7, 11, 96]. Then, it recreates the high-level control and data flow inside each service, and, finally, it generates system calls and user-space assembly to capture the on-CPU and off-CPU behavior. Ditto operates transparently to the user, with the cloning process working in an automated fashion, from obtaining a microservice deployment's dependency graph to populating each tier with appropriate assembly code and I/O operations. It generalizes across platforms, deployments, and application configurations, such as load and thread pool size, without retraining, and the synthetic applications react to changes similarly to the original ones.

Ditto is beneficial to hardware vendors, cloud providers, and researchers. Hardware vendors can obtain synthetic versions of production applications to test new platforms, cloud providers can specify performance and/or resource specs to hardware vendors using the synthetic workloads, and researchers can use representative end-to-end cloud services without the need for production code access.

We evaluate Ditto across a set of both monolithic applications and multi-tier microservices and show that it consistently captures the low- and high-level performance metrics and resource characteristics of the original service. We also validate that synthetic applications generated with Ditto react the same way as the original workloads to changes in the input load, platform, resource allocation, and deployment configuration, including interference from external workloads and power management. Ditto is open-source software. [1]

2 RELATED WORK

2.1 CPU and Cloud Benchmarking

The architecture and system community rely heavily on software benchmarking to learn the performance characteristics of target applications. Prior studies have found that traditional CPU benchmark suites, such as SPEC [32, 55, 99] and MiBench [61], differ significantly from the services running in production clouds [100, 103, 111].

One of the earliest efforts towards modern cloud application benchmarking was the Cloudstone benchmark [97], which proposed a new interaction-heavy Web 2.0 workload. CloudSuite [48] further composes a collection of workloads for the evaluation of scaling-out cloud services. The YCSB suite [37] collects workloads for database systems, while SPEC Cloud [9] utilizes a subset of workloads representing real-world use cases found on IaaS clouds. More recently, uSuite [98] and DeathStarBench [53] focus on benchmarking cloud microservices, given the increased popularity of this programming model.

Instead of condensing cloud services to a pre-set group of benchmarks, Ditto enables generating arbitrary applications that resemble in features a target service.

2.2 Simulation and Trace Replay

Simulation and trace replay provide another way to estimate service performance when hardware or software is inaccessible. Many microarchitectural simulators, including gem5 [31], Sniper [34] and ZSim [94], can accurately simulate the CPU performance of a given binary. BigHouse [82] and μqsim [109] are queueing-based simulators which quickly estimate high-level performance metrics of monolithic applications and microservices. While useful when hardware is not available, these simulators still make approximations about the application behavior, and do not capture all complexities of a real system. On the other hand, RecPlay [92], iDNA [29], PinPLay [89], Jalangi [95] log the execution and memory traces, and reproduce an application's behavior for debugging and performance analysis. Unfortunately, a lot of prior work has showed that traces can leak confidential information about production services [33, 38, 66], restricting an application owner's incentive to publicly share the collected traces. West [22], STM [20], HALO [86] and Dangwal's paper [39], for example, analyze the memory access patterns of an original application, and generate synthetic memory traces. Although they can constrain the information leakage, they only target the cache and memory subsystems. Compared to trace-based techniques, Ditto generates synthetic services that clone the performance characteristics across the system stack, and can run both on real systems and microarchitectural simulators.

2.3 Performance Cloning and Synthetic Benchmarks

Workload cloning is a way to generate synthetic code that mimics real-world applications. Previous studies profile architecture-independent characteristics of real applications, and generate corresponding proxy benchmarks that capture their CPU performance [26, 56, 73, 87]. PerfProx, for example, generates miniature proxies which resemble the low-level CPU metrics of real databases [87]. MicroGrad [90] introduces a gradient-based mechanism to generate workload clones and stress tests. NanoBench [18] generates microbenchmarks with certain instructions to evaluate undocumented features of x86 CPUs. In [104], the authors hide the functional semantics of the proprietary applications through code mutation.

However, these systems are not sufficient for performance cloning in cloud services. First, they only consider performance

[1] https://github.com/Mingyu-Liang/Ditto.

metrics in user space. Cloud applications spend a large fraction of their execution at kernel level [43, 48, 53, 74, 77, 99]. Synthetic benchmarks generated with these tools focus on matching low-level performance metrics, e.g., instructions per cycle (IPC) or misses per kilo instructions (MPKI), which do not always translate to the high-level metrics cloud applications care about, like tail latency and throughput [28, 40]. Second, cloud services are often bottlenecked by off-CPU events, such as context switching or network or disk I/O, which are not captured in previous work. Finally, cloud services do not operate like independent processes, having instead client-server interfaces, which need to be captured by the cloning framework. This is even more the case for multi-tier microservices, which can have hundreds of dependent tiers, and are becoming the norm in many clouds.

3 CLONING ACROSS THE SYSTEM STACK

Application cloning for cloud services is challenging due to the complexity and heterogeneity of their design, and the various platforms they can be deployed on. Different services can have entirely different bottlenecks; for example, key-value stores (KVS) require high CPU performance, high memory and network bandwidth to retrieve a large amount of data under a strict latency SLO, while databases are usually bottlenecked by disk I/O bandwidth [36]. Therefore, it is important to consider the performance breakdown across the system stack to accurately clone the performance of end-to-end cloud services.

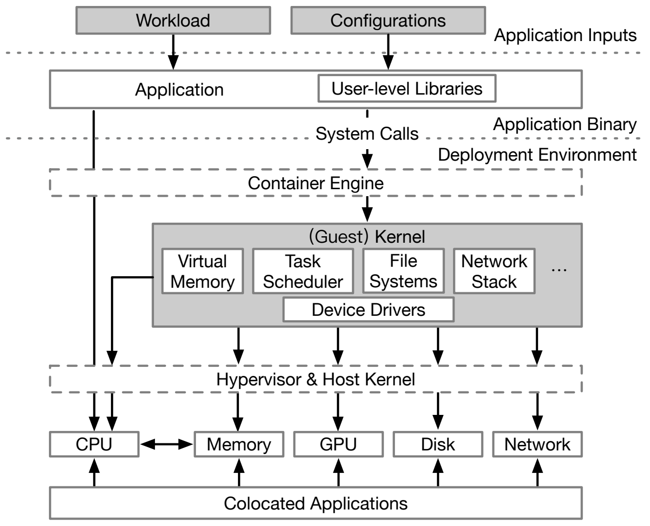


Figure 1: General system stack for cloud applications [59]. Dashed boxes are optional layers for virtualization.

Figure 1 demonstrates an abstract view of a generic system stack for a single cloud server [59]. The performance of an application is determined by factors that range from the application code and inputs, to the environment it is running on, including containerization technology, the hypervisor, server platforms, and any colocated applications. We briefly describe why these factors matter below.

3.1 Application Inputs

The behavior and performance of cloud applications is significantly impacted by the service configuration and input load, with the latter going through well-documented fluctuations [19, 24, 42, 43, 81, 91, 99]. The application's configuration, although changing less frequently than load, can substantially alter the execution flow of an application and impact performance. For instance, configuring a smaller in-memory cache for a database can cause more disk I/O accesses, significantly increasing latency.

3.2 Application Codebase and Binary

The application and its linked libraries are intrinsic to its performance, regardless of the platform it is deployed on. Modifications in the application code can alter the control and data flow of a service, its memory access patterns, and its resource bottlenecks. This is especially true for new cloud programming frameworks, like microservices and serverless, where services are updated on a daily basis.

3.3 Deployment Environment

3.3.1 Containers and Virtual Machines (VMs). Cloud services are often deployed with containers and/or VMs. These add different levels of performance overheads, primarily due to the extra I/O and network layers [47]. Unlike prior work, Ditto faithfully clones the I/O behaviors of the cloud services, and thus, the synthetic applications generated by Ditto can be affected by virtualization the same way as the original services.

3.3.2 OS Kernel. Cloud applications are especially dependent on OS performance, given that they spent a large fraction of their execution at kernel level for interrupt handling, I/O requests, memory management, task scheduling, etc. [25, 53, 74, 76]. Prior work on application cloning has mostly focused on user-level application logic; for cloud services overlooking kernel operations leads to very different performance characteristics compared to the original application.

3.3.3 CPU-Memory Subsystem. The CPU-memory subsystem is a dominant factor in cloud application performance, even for services that spend significant time processing network requests [53, 74, 76]. We follow the top-down analysis methodology in [107] to identify the key CPU performance metrics that impact the overall IPC and reproduce them in the synthetic applications, as shown in Figure 2. Section 4.4 discusses how Ditto accounts for each of these factors during application generation.

3.3.4 Hardware Devices. Services interact with hardware devices, including disks, and NICs through system calls. In cloud services specifically, peripherals can dominate performance, especially when they experience long queueing delays. We mainly consider the impact of storage and network devices in our study, as many cloud services involve I/O and network operations. Ditto can be extended to clone the behavior of other devices, such as GPUs and hardware accelerators, which we defer to future work.

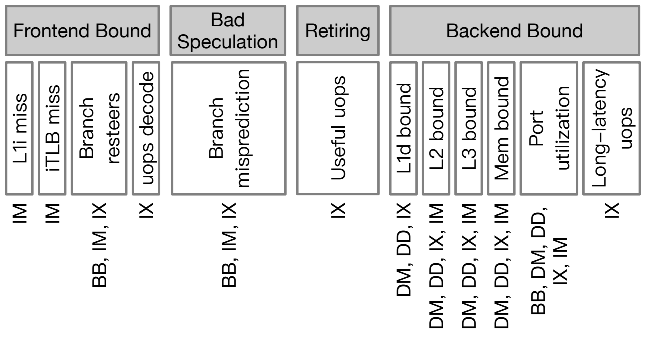


Figure 2: Top-down analysis of the CPU-memory subsystem performance [107]. Letters at the bottom show the corresponding analysis in Ditto. IX: Instruction Mix. BB: Branch Behavior. IM: Instruction Memory Access Pattern. DM: Data Memory Access Pattern. DD: Data Dependency.

3.4 Multi-Tenancy

Multi-tenancy improves datacenter utilization by deploying multiple services on the same node. Applications share resources, including CPU cores, LLC, and memory, disk I/O, and network bandwidth [36, 79]. Resource contention can degrade performance, and should be accounted for in the application cloning process.

4 END-TO-END CLONING FOR CLOUD SERVICES

4.1 Overview

Ditto is an application cloning framework for cloud services; it applies to both single-tier applications and multi-tier microservices. It generates services that faithfully reproduce the performance, resource profile, and thread-level control/data flow of the original workload, decoupling representative system studies from access to the source code or the binary of production cloud services.

Ditto profiles an application at runtime and extracts key performance and resource metrics using dynamic instrumentation and runtime emulators (SystemTap [45], Valgrind [85], eBPF [44], Perf [12], VTune [67], and Intel SDE [68]). Then, it generates a synthetic service which preserves the performance of the original, using an entirely distinct code sequence, to avoid revealing the implementation of the original service.

Figure 3 shows an overview of Ditto's profiling and generation process. If the target service consists of a set of microservices, Ditto first learns their Remote Procedure Call (RPC) dependency graph, using distributed tracing [4, 7, 11, 96]. This graph is then used to generate the API interfaces between the different synthetic microservices. Next, Ditto analyzes the thread and networking model, e.g., single- or multi-threaded, and synchronous or asynchronous respectively using kernel-level profiling, and builds the skeleton of each service. The application skeleton contains empty handlers which are filled with appropriate functionality in the next step. The handlers can either be triggered upon receiving requests for worker threads, or by a timer for background threads.

To generate the synthetic application body, Ditto instruments the application binary using kernel- and user-space profilers for different subsystems. Finally, Ditto uses the deviation in performance metrics between original and synthetic application to fine tune the generator. The eventual synthetic service can serve as a performance and resource proxy for the original service.

Ditto profiles applications in isolation to capture their characteristics alone; in Section 6.5 we show that in the presence of interference, synthetic applications behave the same way as their original counterparts.

Ditto adheres to the following design principles:

- **End-to-end system stack modeling:** Cloud services often contain a large fraction of kernel-space operations for network and disk I/O. Ditto captures the inputs, RPC dependency graph, application binary, OS kernel, CPU, memory, disk, networks, and resource interference.
- **Portability:** Ditto uses platform-independent features to ensure that generated services are portable across platforms without reprofiling. Synthetic applications also faithfully adjust to load and configuration changes, such as queries per second (QPS), and scaling, because of the fine-grained network and thread modeling.
- **Abstraction:** Ditto does not disclose the implementation of the original application, only exposing the skeleton and post-processed performance characteristics to the synthetic benchmark user. It replaces the skeleton of an application with a template, refills the body with artificial instructions and their operands, and abstracts the memory access patterns away to avoid side-channel attacks. Application-specific characteristics, including user-space function calls, memory accesses, and application inputs, are also concealed. Thus, the synthetic workload can be publicly shared, without a user reverse engineering the implementation of the original service.
- **Automation:** Ditto automates the profiling and generation process. It entirely relies on static and dynamic profiling of the original application to generate a benchmark. Users are not required to have expertise in the implementation of a service to use the framework.

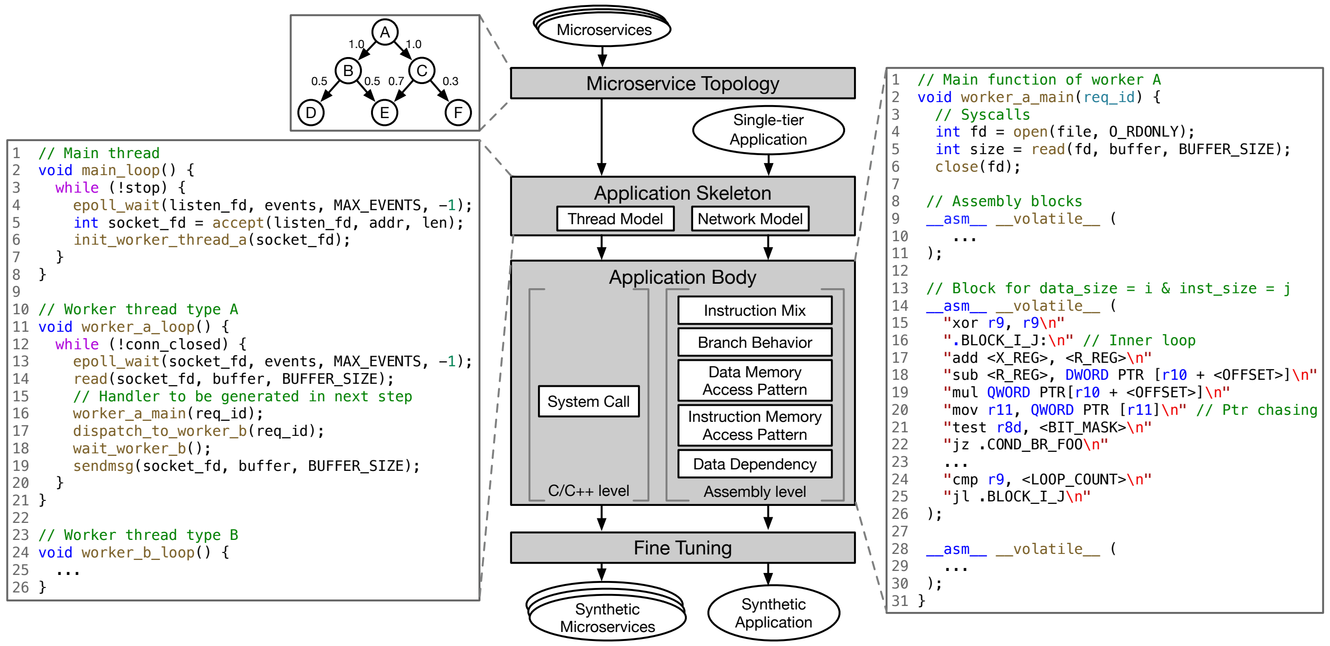
4.2 Microservice Topology

A topology of microservices is a directed acyclic graph (DAG), where the nodes are microservices and the edges indicate the dataflow between dependent tiers [52–54, 80]. Ditto leverages the distributed trace frameworks present in most production deployments to collect traces of end-to-end requests. The performance overhead is negligible if the traces are sampled properly [4, 11, 96]. It then automatically extracts the dependency graph between microservices and uses it as input to the skeleton generator.

4.3 Application Skeleton

We define the application skeleton as the network and thread models of an application, which determine how it handles remote service communication, and how tasks are assigned to different threads, respectively. The application skeleton is a critical design choice for cloud services facing tight latency constraints [46, 88, 101, 106], as it directly impacts their performance and scalability.

4.3.1 Network Model. The network model describes how an application communicates with other services, acting as a client, server,



```
// Main thread
void main_loop() {
  while (!stop) {
    epoll_wait(listen_fd, events, MAX_EVENTS, -1);
    int socket_fd = accept(listen_fd, addr, len);
    init_worker_thread_a(socket_fd);
  }
}

// Worker thread type A
void worker_a_loop() {
  while (!conn_closed) {
    epoll_wait(socket_fd, events, MAX_EVENTS, -1);
    read(socket_fd, buffer, BUFFER_SIZE);
    // Handler to be generated in next step
    worker_a_main(req_id);
    dispatch_to_worker_b(req_id);
    wait_worker_b();
    sendmsg(socket_fd, buffer, BUFFER_SIZE);
  }
}

// Worker thread type B
void worker_b_loop() {
  ...
}
```

```
// Main function of worker A
void worker_a_main(req_id) {
  // Syscalls
  int fd = open(file, O_RDONLY);
  int size = read(fd, buffer, BUFFER_SIZE);
  close(fd);

 // Assembly blocks
 __asm__ __volatile__ (
    ...
 );

 // Block for data_size = i & inst_size = j
 __asm__ __volatile__ (
   "xor r9, r9\n"
   ".BLOCK_I_J:\n" // Inner loop
   "add <X_REG>, <R_REG>\n"
   "sub <R_REG>, DWORD PTR [r10 + <OFFSET>]\n"
   "mul QWORD PTR[r10 + <OFFSET>]\n"
   "mov r11, QWORD PTR [r11]\n" // Ptr chasing
   "test r8d, <BIT_MASK>\n"
   "jz .COND_BR_FOO\n"
   ...
   "cmp r9, <LOOP_COUNT>\n"
   "jl .BLOCK_I_J\n"
 );

 __asm__ __volatile__ (
   ...
 );
}
```

Figure 3: Overview of Ditto's synthetic benchmark generation process.

or both. When acting as a client, a service can use synchronous or asynchronous communication. In synchronous models, threads block on network I/O (e.g., `send()`, `write()`) to await responses. Asynchronous models are typically event-based with responses handled by specific threads via callback functions. They are more complicated, as they involve additional synchronization and state machine transitions. In return, they avoid long queueing delays by allowing threads to process new requests and offer better performance [101].

On the server side, there are three common options for the network model: blocking, non-blocking, and I/O multiplexing [102]. In all three models, threads await requests through system calls (e.g., `recv()`, `read()`, `epoll()`). In contrast to the other two models, the non-blocking model needs to periodically call the I/O interfaces to look for new requests, which can waste CPU time at low loads. In both blocking and I/O multiplexing models, threads block on system calls, although I/O multiplexing allows monitoring multiple sockets via a single system call (e.g., `select()` or `epoll()`). I/O multiplexing is the most commonly-used in services like Memcached, Redis, and NGINX, since they support many concurrent connections, and I/O multiplexing reduces the required threads.

Ditto uses SystemTap [45] to profile the network model by probing kernel-space functions and data structures. It acquires key attributes of sockets, and monitors network-related system calls, gathering the distribution of their types, arguments, and call frequency. Ditto then chooses one out of several network models that combine the different design choices described above, with socket options and network message parameters set based on profiling.

4.3.2 Thread Model. Cloud services rely on multithreading for asynchronous networking, disk I/O, and parallel processing [111]. The thread model describes how tasks are scheduled to and handled by various threads. Ditto uses SystemTap to profile the functionality, lifecycle, and trigger points of threads by experimenting with different connections, QPS, and execution times. First, it combines network and user-space call stack analysis to cluster threads with similar functionalities. We build a call graph for each thread, use tree-edit distance [30] to measure the similarity between threads, and cluster threads with similar call graphs using agglomerative clustering [83], since the number of clusters is unknown in advance. Second, we categorize each thread cluster into short- and long-lived threads by probing `clone()` and context switches. Short-lived threads are usually spawned and terminated frequently, while long-lived threads are spawned at initialization, waiting for tasks to arrive. Finally, thread functions can be triggered by both kernel- and user-space events, including reads and writes to sockets, timers, signals, user-space locks, and condition variables. We monitor event notification functions in kernel space and common user-level libraries, such as `libpthread` and `libc++`, and analyze the relationship between them and thread spawning or wakeup to identify trigger points.

4.4 Application Body

The application body corresponds to the workload-specific work, consisting of kernel-space functions, via system calls and user-level functions. While assembly-level profiling for kernel-space functions is unnecessary, since they can be cloned by imitating the system calls themselves, it is critical to clone user-space functions at assembly level to capture the low-level usage of CPU resources.

Application performance is also significantly impacted by factors like instruction mix, and memory (data and instruction) access patterns, branch behavior, and data dependencies. Ditto uses these platform-independent features to ensure that the generated synthetic applications can be ported to other platforms without reprofiling.

4.4.1 System Calls. Applications use system calls to perform privileged operations in the OS kernel. Besides network handling and spawning new threads, cloud applications can make system calls to access file descriptors, allocate memory space, or synchronize on shared memory. Capturing the system call characteristics is critical to clone the kernel-level CPU and un-core metrics. Prior performance cloning studies either do not profile system call characteristics [72, 90], or only profile the total number of kernel-level instructions [87]. To accurately capture kernel-level characteristics, Ditto profiles the distribution of system calls, including their counts and arguments with SystemTap. For example, MongoDB calls `pread()` to read a database file from disk. During system call profiling, Ditto captures the flags of `fd` and the distribution of `count` and `offset`, to accurately clone key metrics, such as disk latency, utilization, and page cache miss rates.

4.4.2 Instruction Mix. The instruction mix in Ditto captures the distribution of x86 assembly instructions at runtime in the original service, and reproduces it faithfully in the synthetic benchmark. Previous studies categorize x86 assembly instructions into integer arithmetic, integer multiplication, integer division, floating-point operations, SIMD operations, loads, stores, and control instructions [72, 87, 90]. They then generate the synthetic benchmark using a representative instruction from each category.

However, this categorization is too coarse-grained and does not capture the characteristics of modern CPU microarchitectures. The x86 ISA, for instance, contains assembly instructions with different uops, port usages, and execution cycles. For example, the `CRC32 (r64, r64)` instruction, which implements the checksum function, takes three cycles and can only be executed via port 1 on Skylake CPUs, while other integer arithmetic instructions usually take one cycle on any of the ports 0, 1, 5, and 6 [17, 60]. Instructions with `REP`/`REPZ`/`REPNZ` (repeat string operations) or `LOCK` prefixes can take tens of cycles or more, depending on the repeat count, or the cache/RAM configuration [51].

Ditto uses Intel SDE [68] to collect the dynamic count of each x86 instruction using Intel x86 Encoder Decoder (XED) Iforms [35]. It then clusters x86 assembly instructions by functionality (data movement, arithmetic/logic, control-flow, lock-prefixed, and repeat string operations), operands (general-purpose registers, x87 floating-point registers, XMM registers, and memory), and ALU usage [17] using hierarchical clustering, so that each cluster has similar hardware resource requirements. Ditto also profiles the average number of dynamic instructions per request, and the repeat counts of each `REP`-prefixed instruction. During the generation phase, Ditto randomly samples the next instruction from the instruction mix distribution. Registers and memory addresses are assigned after data memory access profiling (Section 4.4.4).

4.4.3 Branch Behavior. Branch prediction accuracy, which is determined by both the branch behavior of the application and the branch predictors, is critical in modern out-of-order CPUs [60, 78]. Prior studies observe that branch taken ratios and transition rates (frequency a branch switches between taken and not-taken) impact the branch prediction accuracy and misprediction penalty [50, 64]. Branches with extremely high taken or not-taken ratios, even if their patterns are completely random, have fewer mispredictions, since the majority of executions are in one direction. Similarly, branches with low transition rates are easier to predict. We also find that instruction locality and the number of static branch instructions significantly contribute to the branch prediction accuracy, especially for applications with large binaries.

Based on these observations, Ditto profiles the distribution of branch taken/not-taken rates and transition rates across all conditional branch instructions, and together with the instruction memory access pattern analysis it accurately clones the branch misprediction behavior of the target application. We quantize the taken/not-taken rates and transition rates in log scale, from 2^{-1} to 2^{-10}. During the generation phase, Ditto samples a taken/not-taken rate and transition rate from the profiled distribution for each conditional branch instruction.

Lines 21-22 in the right code snippet in Figure 3 show how Ditto generates conditional branch instructions with profiled taken/not-taken rates and transition rates. `<BIT_MASK>` is a binary mask precomputed during the generation phase, which contains M ones in the highest bits and N zeros in the lowest bits. 2^{-M} is the taken/not-taken rate, and 2^{-N} is the transition rate. The `ZF` flag, which determines the branch direction of `jz` or `jnz`, will change periodically according to the bitmask in the `test` instruction.

4.4.4 Data Memory Access Pattern. The memory access pattern is a dominant characteristic of an application, as it impacts the backend of the CPU and memory subsystem. Since operands in arithmetic instructions in synthetic benchmarks are randomly generated, they cannot calculate meaningful memory addresses at runtime. Thus, memory addresses or offsets need to be pre-calculated in the generation phase and hard-coded in the synthetic application binaries. Previous studies [20, 22, 86] capture memory access patterns using the stack distance, reuse distance, and stride pattern profiles. However, they need 10 to 20 million memory traces to accurately represent target memory access patterns because of the sparsity of the memory address space, and the multimodality in memory accesses [63]. Preserving the original access patterns requires millions of hard-coded memory instructions, which significantly interferes with other performance characteristics. Moreover, directly replicating the target memory access pattern introduces security concerns, since previous studies showed that memory access patterns reveal confidential information about the service [57, 66, 71].

Instead, Ditto uses profiling of the memory working set to synthesize appropriate data memory access patterns without incurring high instruction misses or leaking application context. We construct a sequence of memory accesses for working sets with different sizes, from 64 bytes (one cache line) to the maximum memory size allocated to the target application, increasing by a factor of two. Each memory access only reads or writes the first data in a cache line to ensure that a new cache line is loaded, assuming the most common write-allocate policy. We use Valgrind [85] to compute the distribution of memory accesses with different working set sizes, which can be efficiently simulated as "cache hits" for different "cache sizes". Each "cache size" only needs to be simulated once during profiling. We calculate the number of memory accesses in a working set of 2^i bytes as follows:

$$A_d(2^i) = \begin{cases} H_d(2^i) & \text{if } 2^i = 64 \text{ bytes} \\ H_d(2^i) - H_d(2^{i-1}) & \text{otherwise} \end{cases}, \quad (1)$$

where $A_d(2^i)$ is the number of memory accesses for a working set of 2^i bytes in generated code, and $H_d(2^i)$ is the number of cache hits in a 2^i-byte cache in the original application. The synthetic working set-based memory access pattern is illustrated in Figure 4, with the number of memory accesses for each working set equal that of the profiled distribution. Since the memory accesses are limited to the working set size, it is guaranteed that $A_d(2^i)$ accesses will contribute to $A_d(2^i)$ hits when cache size $\geq 2^i$ bytes. Assuming a least-recently-used (LRU) cache replacement policy or its pseudo-LRU variant, commonly used in recent Intel processors [18, 105], since we iterate through cache lines in a working set sequentially, there must be previous memory accesses which evict this cache line when cache size $< 2^i$ bytes. Therefore, every memory access of a 2^i-byte working set ends up with a miss when cache size $< 2^i$ bytes. The statement is true for any memory hierarchy and cache inclusion policy because of the sequential access pattern within each working set. Therefore, even if applications are profiled with a single-level cache, the results can be applied to any number of cache levels and inclusion policies. Applications are profiled with an 8-way cache for working sets < 1MB and a 16-way cache for working sets ≥ 1MB, which are close to the typical values of modern CPUs. There is an average 1.9% error in the cache miss rate when cache associativity changes across all examined applications. We allocate an array for memory accesses in the heap when the synthetic application is initialized, and store the base address in a register (for example, `r10`). Ditto generates the address offsets for each memory instruction, which can access `[r10 + <OFFSET>]` at runtime.

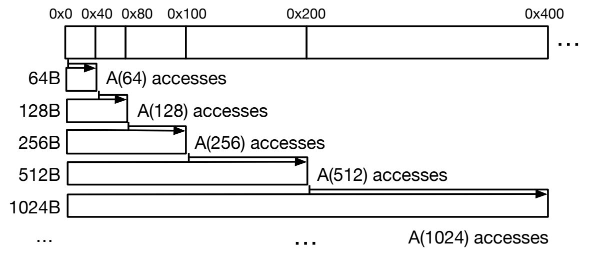


Figure 4: Working-set-based data memory access generation. Except for the 64-byte working set, the memory accesses of 2^i-byte working set start at address 2^{i-1} and loop iteratively within the working set.

Coherence misses also contribute to cache miss rates in multi-threaded applications. Coherence misses happen when cache lines containing shared data are invalidated by another core. To accurately clone cache behavior with multi-threading, we use Intel SDE to profile the ratio between private data accesses and shared data accesses, and generate memory accesses accordingly.

Modern CPUs implement hardware prefetching mechanisms to improve cache performance. Hardware prefetchers detect load instructions with regular strides, sequences of consecutive cache line accesses and adjacent cache line accesses, to load data into caches before they are needed [84]. To clone the performance impact of cache prefetching, we calculate the ratio of regular to irregular memory access patterns from the runtime memory trace and use this ratio to control the number of regular memory access sequences in the synthetic applications.

4.4.5 Instruction Memory Access Pattern. Instruction memory access patterns significantly impact CPU frontend and backend performance, as they determine the L1i, L2, L3 cache misses and branch mispredictions. Replicating the original application's instruction memory access pattern is not possible with a synthetic benchmark because the execution flow is usually controlled by the computation's output at runtime.

Therefore, Ditto synthesizes instruction memory access patterns with a similar approach to that of Section 4.4.4. We profile the i-cache hits of the original application with different i-cache sizes using Valgrind. Then, we calculate the distribution of dynamic executions in an instruction memory working set of 2^j bytes as follows, assuming the cache line size is 64 bytes, and the average instruction size is 4 bytes:

$$E_i(2^j) = \begin{cases} 16 * \left[H_i(2^j) - H_i(2^{j-1})\right] & \text{if } 2^j > 64 \text{ bytes} \\ H_i(2^N) - \sum_{j=9}^{2^N} E_i(2^j) & \text{if } 2^j = 64 \text{ bytes} \end{cases}, \quad (2)$$

where $E_i(2^j)$ is the number of instruction executions with a working set of 2^j bytes in the synthetic code, 2^N is the max instruction working set size, $H_i(2^j)$ is the number of i-cache hits on a 2^j-byte i-cache in the original application, and the number of instructions in a cache line is 16 (64B cacheline / 4B inst size). After profiling the distribution of i-cache accesses with different instruction working sets, Ditto generates static assembly instruction blocks, shown in lines 14-26 in the right code snippet of Fig. 3.

The number of instructions per block matches the instruction working set size, and the loop iteration number is determined by the distribution.

4.4.6 Data Dependencies. Data dependencies are another inherent characteristic of an application that impact performance. Data dependencies can flow through registers or memory locations, limiting the number of simultaneous instructions issued to an execution unit (instruction-level parallelism, or ILP), and the number of outstanding memory requests (memory-level parallelism, or MLP) [65].

Ditto uses the distribution of data dependency distances to quantify data flows through registers. We measure the read after write (RAW), write after read (WAR), and write after write (WAW) data dependency distance from the dynamic control flow graph (DCFG) generated using Intel SDE [69]. The dependency distance is quantized into 11 bins, increasing exponentially from 1 to 1024, since a larger dependency distance does not impact the ILP, due to the limited size of the reorder buffer. When generating the synthetic code, we reserve several registers for recording the loop counters and data memory addresses, and use the rest of general-purpose and SIMD registers to clone the data dependency characteristics. To assign registers for each instruction, Ditto samples a (RAW, WAR, WAW) distance tuple from the profiled distributions, and chooses an available register with the closest distance values. Data dependencies through registers can also impact MLP if the register values determine memory locations. Such behavior cannot be captured since the synthetic application never writes to a reserved register with the memory base address. To address this, we replace a fraction of memory reads with pointer chasing reads (`mov r11, QWORD PTR [r11]`); determined by the MLP measured with Perf.

Data dependencies through memory locations are much more difficult to profile with DCFG since memory addresses are often calculated at runtime. However, they are partially determined by data access patterns that Ditto already profiles (Section 4.4.4). A program with a shorter memory dependency distance can be modeled with a smaller working set because the probability that the cache line is evicted by other instructions in between is lower.

4.5 Fine Tuning

Finally, Ditto implements fine tuning to calibrate the output of previous steps, due to inaccuracies introduced by the instrumentation tools. For example, application body profiling does not consider the interaction between user-space and kernel-space functions and the correlation between the application skeleton and body; thus the actual d-cache and i-cache miss rates are often higher than the profiled results. Ditto iteratively runs the synthetic application on a specific platform, computes the errors between target and synthetic service, adjusts the inputs to the generator accordingly, and regenerates the synthetic application. Although there are many knobs to tune, most of them are orthogonal with each other. We have characterized the correlation across knobs, and derived the small groups of parameters that need to be jointly tuned (e.g., branch taken/transition rate and i-cache pattern because they all influence branch prediction). Since relationships between knobs and performance are mostly linear, we use a feedback-based heuristic to tune knobs within a group. Fine tuning uses performance counters for calibration. It usually takes within ten iterations to reach over 95% accuracy, incurring low overhead since each iteration only takes a couple tens of seconds. Since Ditto captures performance characteristics well with platform-independent data, this fine tuning does not compromise the generality of the synthetic service, as shown in Section 6.2.2.

5 IMPLEMENTATION

Ditto implements several analyzers and code generators to capture the microservice topology, application skeleton, and application body. If the target service is a graph of microservices, the microservice topology analyzer leverages distributed tracing systems, like Jaeger [4], to obtain RPC call graphs and call statistics. For both microservices and single-tier applications, the application skeleton analyzer then deploys SystemTap to profile network- and thread-related functions and data structures in kernel space, and identify the network and thread models used.

The skeleton generator creates a synthetic application skeleton using either a TCP- or RPC-based network interface, leaving the body of each thread's handler to the application body generator. The latter runs SystemTap to profile system calls, and uses Intel SDE and Valgrind to capture the platform-independent features of binaries, such as instruction mix and working set size distribution, etc. The generator creates handlers according to these features using POSIX APIs in libc and inline assembly in C code. The assembly code contains tens to hundreds of instruction blocks looping iteratively with different instruction and working set sizes. Finally the fine tuner runs the synthetic application, collects performance data from Perf, eBPF and VTune on the deviation between original and synthetic workloads, and calibrates the input data for the application body generator accordingly.

Ditto is implemented primarily in Python and C in about 16,000 lines of code. It supports C/C++ applications, the Apache Thrift [1] and gRPC [3] RPC frameworks, and x86 ISAs, which are commonly used in cloud environments. It can be extended to more languages, frameworks, and ISAs, by leveraging compatible profiling tools. Ditto can generate applications that run on a single machine or containerized microservices that run on multiple nodes, using Docker Swarm or Kubernetes. The runtime profilers and emulators, including SystemTap, Intel SDE, and Valgrind, can introduce overheads to the original application during profiling. This overhead only occurs once, and does not affect the accuracy of the platform-independent features collected during profiling.

To generate a clone, cloud providers only need to specify a representative input for their service. Ditto automatically instruments the application at runtime, collecting profiling statistics and feeding them to the code generator, followed by the fine-tuning process. Ditto does not require reprofiling if the input change does not affect the application body, such as changes in QPS or number of connections. Inevitably, if a new input exercises an entirely new code path or memory access pattern, this will need to be profiled to create a new clone. The synthesized binaries can run directly on hardware, execution-driven simulators like gem5 [31] and ZSim [94], or their traces can be fed to trace-driven simulators like Ramulator [75].

6 EVALUATION

6.1 Methodology

6.1.1 Platforms. We validate Ditto on a heterogeneous cluster, with three types of servers, whose specs are in Table 1. All servers run x86 ISA, but differ in the CPU and memory architectures, and their storage and network.

Table 1: Server platform specifications.

	Platform A	Platform B	Platform C
CPU model	Gold 6152	E5-2660 v3	E3-1240 v5
Base Frequency	2.10GHz	2.60GHz	3.50GHz
CPU cores	22	10	4
CPU family	Skylake	Haswell	Skylake
Sockets	2	2	1
L1i/L1d	32KB/32KB	32KB/32KB	32KB/32KB
L2	1MB	256KB	256KB
LLC	30.25MB	25MB	8MB
RAM	192GB@2666	128GB@2400	32GB@2133
Disk	1TB SSD	2TB HDD	1TB HDD
Network	10Gbe	1Gbe	1Gbe

6.1.2 Applications and Workload Generators.

- **Memcached:** Memcached [49] is a distributed low-latency, key-value store for in-memory caching. We build Memcached 1.6.9 from source, deployed with four worker threads, and load it with 10K items, each with a 30B key and a 4KB value. It is driven by an open-loop version of the mutated workload generator [13].
- **NGINX:** NGINX [6] is a high-performance web server and is the most commonly-deployed technology in Docker [15]. We build NGINX 1.20.0 from source and configure it with one worker process. For NGINX, we use tcpkali [14] to generate HTTP requests.
- **MongoDB:** MongoDB [5] is an open-sourced cross-platform NoSQL database. We use MongoDB 4.4.4 and set up a dataset

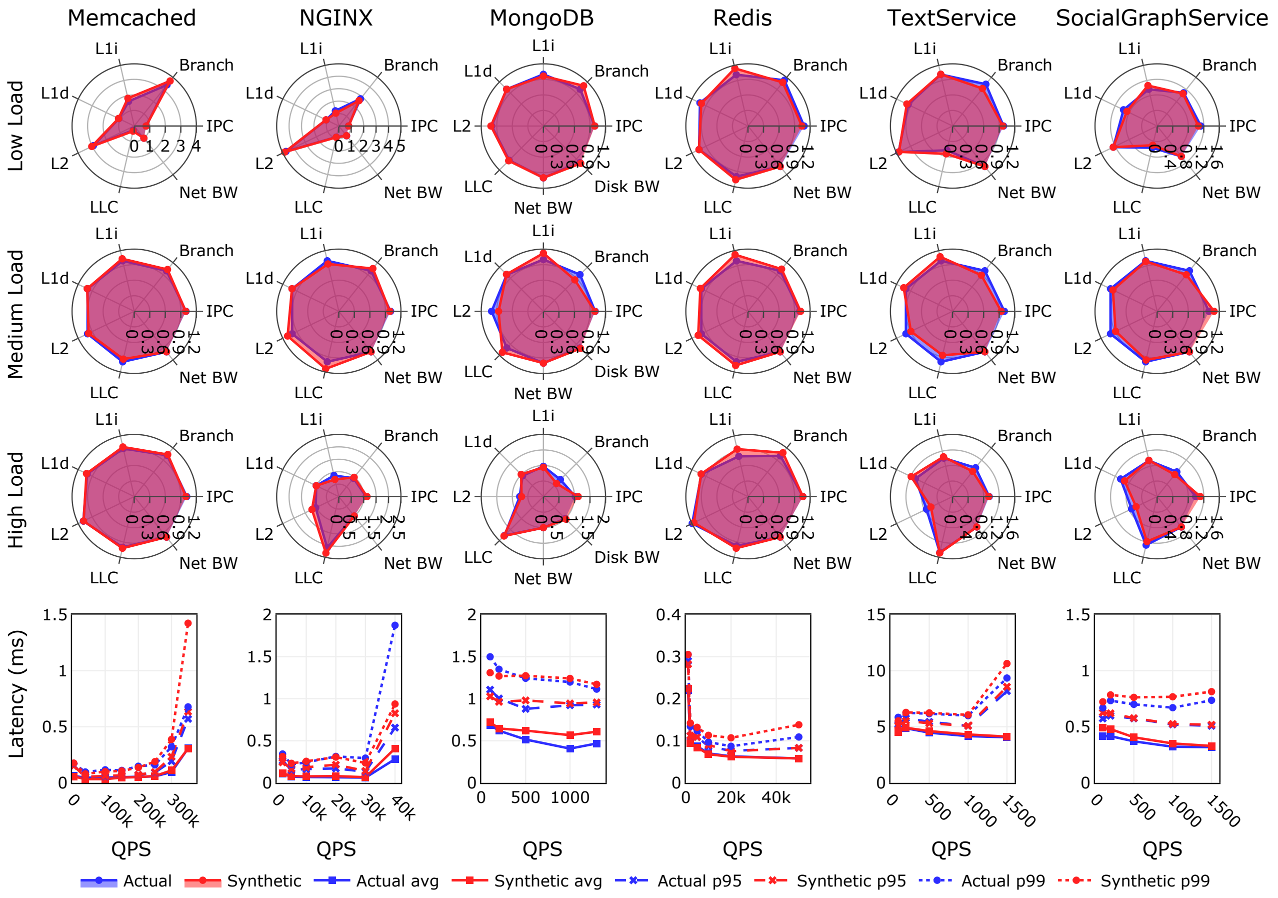


Figure 5: CPU performance metrics (IPC, branch mispredictions, L1i, L1d, L2 and LLC miss rates), network bandwidth, disk bandwidth (MongoDB only) and service latency under varying load across six services. CPU metrics are normalized to each original application's metrics under medium load. Network and disk bandwidth are, by exception, normalized to each original application's bandwidth under current load, because their magnitudes change significantly, and would obscure the figure's shape.

of 40GB with one million records. To load MongoDB, we use YCSB [37] with all read operations, following a uniform distribution.

- **Redis:** Redis [8] is a fast, single-threaded, in-memory data store used as a database, cache, and message broker. We build Redis 6.2.6 from source, disable its persistent storage, and load a dataset with 100K records. We use YCSB as the load generator.
- **Social Network:** Social Network is a microservice topology from DeathStarBench [53], consisting of 20+ individual services. We compose its social graph with the socfb-Reed98 Facebook dataset [93], which contains 962 users and 18.8K follow relationships. We also modify the wrk2 [2] workload generator to open-loop and use it as the client. The Social Network is deployed with one replica per microservice, both locally and on a cluster using Docker containers.

For all synthetic applications, we use the same load generator as the original application, sending dummy requests with the same traffic distribution. The number of threads of MongoDB and Social Network microservices changes dynamically with the number of concurrent connections, up to a few tens under our load settings.

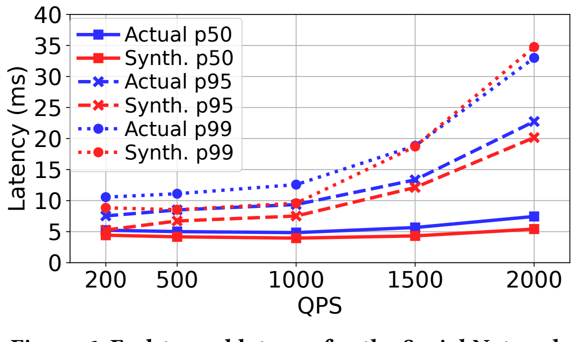


Figure 6: End-to-end latency for the Social Network.

6.2 Validation

6.2.1 Validation on Varying Loads. Figure 5 shows CPU, network and disk performance metrics, and latency for six applications under different QPS in platform A. In addition to the four single-tier applications, we also show resource characteristics for TextService and SocialGraphService, two of the Social Network's tiers, which

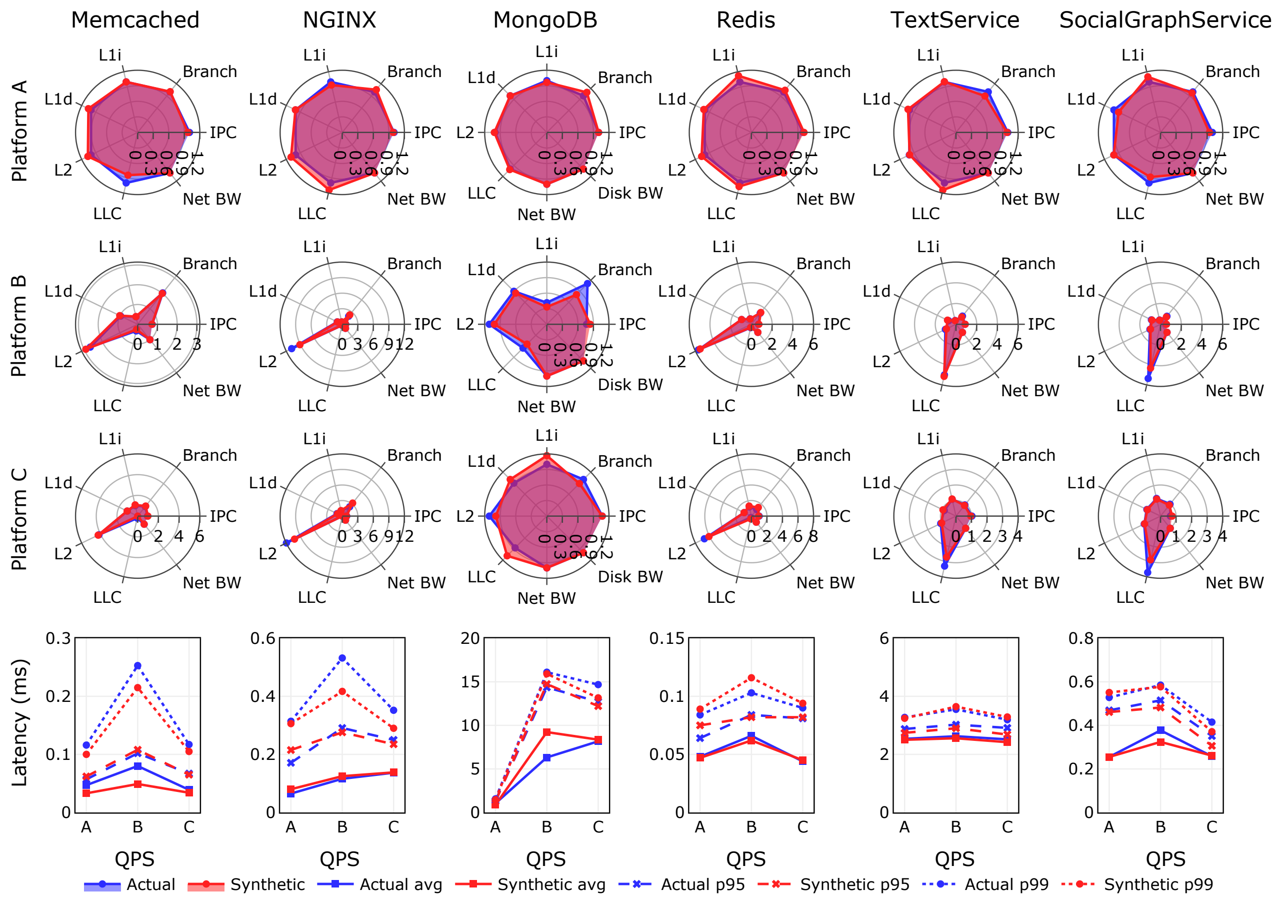


Figure 7: CPU metrics (IPC, branch misprediction, L1i, L1d, L2 and LLC misses), network BW, disk BW (MongoDB only) and latencies across platforms. CPU metrics are normalized to each original service on Platform A.

are representative of the other tiers of the service. TextService manages the text users add to composed posts, and SocialGraphService manages follow relationships between users. We do not show each tier due to space constraints, but have validated that the results are similar for them. All applications are generated using profiling data under medium load; *Ditto has not profiled any other load.* We increase the load until the single-tier application or bottleneck tier in the microservice topology saturates in one or more resources (e.g., disk I/O for MongoDB and CPU for the other applications). Since we use a close-loop workload generator for MongoDB and Redis, which only allows one outstanding request per connection, the latency does not increase significantly at high load. While the end-to-end latency of Social Network increases at high load, the latency of TextService and SocialGraphService only increases slightly, since they are not bottleneck tiers.

The upper three rows show IPC, branch misprediction, L1i, L1d, L2, LLC miss rates, and network and disk I/O bandwidth under low, medium, and high load, with average errors across all applications being 4.1%, 9.9%, 7.1%, 5.1%, 6.9%, 12.1%, 0.1%, 0.1%, respectively. This indicates that Ditto accurately clones the overall hardware performance metrics. Memcached and NGINX have low IPC under low load because of high branch misprediction, and L1i and L2 misses, while SocialGraphService has high IPC due to fewer LLC misses. At high load, Memcached and Redis have similar metrics to medium load, however the other four applications exhibit different degrees of L2, LLC misses variation. The results illustrate that applications can have very different characteristics under different loads, which are accurately captured by Ditto in their synthetic counterparts. The network and disk bandwidth also conform to the original by faithfully reproducing the system calls. We only show disk bandwidth for MongoDB since other services do not involve disk I/O. The bottom line plot shows the average, 95th, and 99th percentile latencies, which also match the originals, with the p99 diverging at high load, due to the queueing behavior in the network stack at saturation.

Fig. 6 shows the end-to-end latency of original and synthetic Social Network when every individual microservice is replaced with a synthetic one.

6.2.2 *Validation on Varying Platforms.*

We also validate the CPU, network and disk metrics and service latency as we vary the hosted platforms. Each application is profiled only on Platform A, and validated on Platforms A, B and C. Figure 7 shows that the synthetic benchmarks react to platform changes in a similar way to

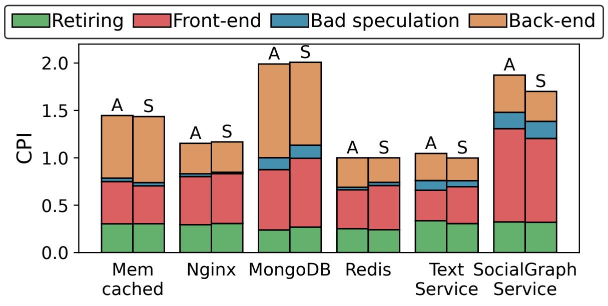


Figure 8: Cycles breakdown. (A: actual, S: synthetic)

the original applications. More specifically, all six applications have different degrees of L2 cache miss increases on Platforms B and C, due to their smaller L2 cache sizes. Applications running on Platform B, which is an older CPU generation, have consistently lower IPC. When running all microservices of the Social Network on the small-scale Platform C server, the high degree of interference results in high LLC miss rates for TextService and SocialGraphService, both original and synthetic. Network and disk I/O bandwidths are identical across platforms, since the amount of data transferred is independent of the platform. The line plots at the bottom show the latency on the three platforms, where the synthetic always matches the original. All applications experience the highest latency on Platform B because it has the lowest IPC. The latency of MongoDB is significantly lower on Platform A because it benefits from the low random access latency of SSDs. In general, the fact that the synthetic applications react to platform changes the same way as the original, without reprofiling, shows that Ditto accurately captures critical, platform-independent features that impact performance.

6.3 CPU Top-down Analysis

Figure 8 shows the cycles per instruction (CPI) top-down analysis of the original and synthetic applications. Ditto accurately captures the cycle breakdown of the original applications. Many prior studies have showed that cloud services diverge from traditional scientific CPU benchmarks like SPEC CPU by having significant fractions of front-end stalls, due to large code footprints and frequent context switches between user and kernel mode [53, 101, 111]. Our synthetic benchmarks show similar bottlenecks to the original applications, and can be used as proxies for microarchitectural optimizations.

6.4 Decomposition of Ditto's Accuracy

In Fig. 9, we use MongoDB as an example to show Ditto's accuracy, as the framework incorporates more information. We start with a version of Ditto that only generates the thread model and network interfaces skeleton, but an empty request handling body. From A to B, we inject the system calls with arguments drawn from the distribution of the original application, which increases the kernel-level instructions and disk I/Os. In C, we add user-level instructions (`add rax, rax`) to match the total instruction count, but not their specific mix. From C to D, user-level instructions are generated based on the profiled mix. We assume the highest branch taken/transition rate, strongest data dependencies, and all memory operations accessing the smallest working sets. We observe an IPC

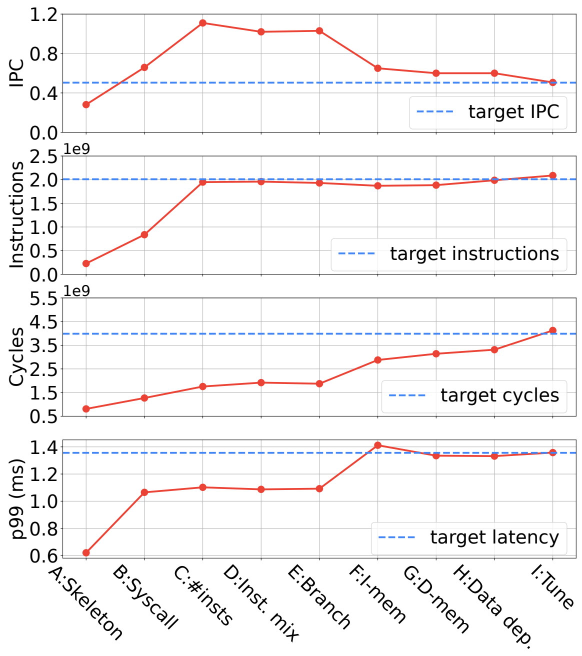


Figure 9: Evolution of IPC, instructions, cycles, and p99 latency for MongoDB as we add sophistication to Ditto.

decrease from 1.11 to 1.02 due to memory instructions incurring additional cycles in the backend. From D to E, we clone the branch behaviors following the profiled branch taken and transition rates. The branch misprediction rate drops from 1.95% to 1.47% but has a negligible impact on IPC. In step F, we synthesize the instruction memory accesses, which causes more i-cache misses (from 1.3% to 7.3%) and branch mispredictions (from 1.47% to 4.56%, as discussed in Sec. 4.4.3), and significantly lowers the IPC. From F to G, we synthesize the data memory access pattern by accessing different sizes of private and shared working sets. The IPC further decreases as the L1d miss rate rises from 17% to 24%. In H, we mimic data dependencies by reassigning registers for each instruction, which clones the ILP and MLP characteristics and slightly lowers the IPC. From H to I, we perform the fine tuning, which calibrates instruction and data access patterns, lowers the IPC from 0.6 to 0.51, and further improves accuracy. This shows that, even if not every aspect in Ditto is equally important, they are all required to accurately clone complex cloud services.

6.5 Case Study: Interference Analysis

Figure 10 shows that synthetic applications react to resource interference in a similar way to their original counterparts, even though we only profile the original application in isolation. We show the analysis on NGINX, but the results are similar for other services. We use a set of stress benchmarks to generate interference in different resources. We use stress-ng [10] to generate hyperthreading (HT), L1d, and L2 interference by co-locating the applications and microbenchmarks on different logical cores of the same physical core. The synthetic application captures the IPC and latency degradation caused by memory contention. When generating L2 interference,

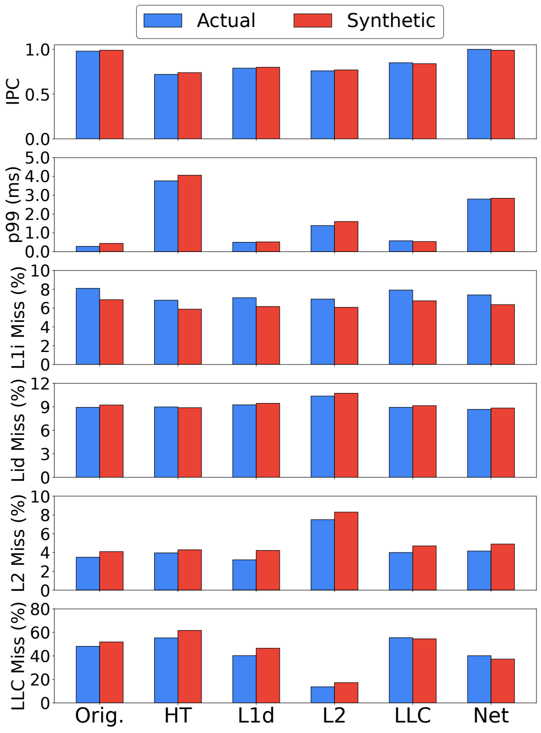


Figure 10: Interference impact on NGINX.

besides the L2 miss rate increase, the synthetic workload also captures the LLC miss rate change in the original service due to an increase in the LLC accesses with constant misses.

We also use iBench [41] to generate LLC interference on the shared socket, and the result shows the synthetic application captures the IPC drop in the original service. Finally, we use iperf3 [16] to compete with the service for network bandwidth, and the latency of synthetic application successfully matches the original service.

6.6 Case Study: CPU Core and Frequency Scaling

Fig. 11 shows using Ditto to evaluate power management in Memcached with CPU core and frequency scaling. Each cell represents the p99 latency under a given number of cores and frequency. We set the QoS as 1ms and cells with marks mean that QoS cannot be satisfied for that configuration. Memcached cannot meet the QoS at low frequency even with the maximum number of cores, which prohibits aggressive power management. Synthetic Memcached accurately captures the latency variation of Memcached under different settings. This similarity indicates that cloud providers can use synthetic applications to determine whether power management is beneficial for a service, without needing access its source code.

7 DISCUSSION

7.1 Suitable and Unsuitable Use Cases

Ditto's main contribution is cloning an end-to-end application across the system stack. This makes it more suitable for architecture-, OS-, application-, and cluster-level studies, including scalability,

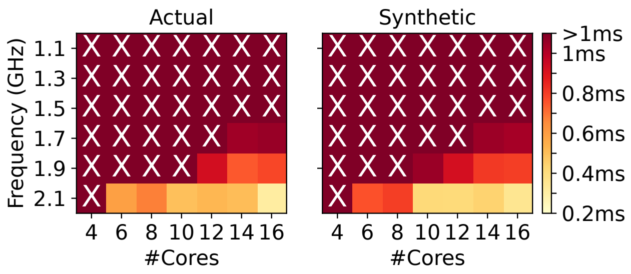


Figure 11: 99th percentile latency of actual and synthetic Memcached under varying CPU frequency core count.

networking, threading, interference, and power management. It can also be used for certain microarchitecture studies, as we showed when changing memory hierarchies across platforms.

However, to enable fast, automated, and obfuscated cloning, our method abstracts away the original individual instructions and memory accesses. Thus, it is less accurate for microarchitecture studies that rely on exact application implementation rather than their statistical performance patterns. For instance, studying the memory access patterns to improve hardware prefetchers would not be a good fit for Ditto. There is a fundamental trade-off between the granularity at which information is captured in one subsystem, and the overall performance accuracy.

7.2 Confidentiality

Although Ditto cannot guarantee zero information leakage, as it may expose the RPC graph, and statistics of some hardware counters, many software companies do not consider these sensitive data. For example, Alibaba, has open-sourced their production RPC traces [80], Facebook shared the kernel-level cycles breakdown of its production workloads [99], and Google open-sourced their workload traces via DynamoRIO [21]. Additionally, the application skeleton, while may reflect the original workflow to some extent, is chosen and adapted from the network and threading models that have been extensively studied and used [46, 101, 108]. As the actual logic, functionality and per-access memory patterns are concealed, inferring useful proprietary information would be very difficult.

For collaboration with hardware vendors, sharing the proprietary code under NDA is of course a more straightforward way, but most cloud providers would not share the binaries related to their core business regardless of NDA agreements. Other solutions like internal evaluation of prototypes can be time-consuming and inaccurate since the prototype is usually immature, and cloud providers need to adapt their workloads for each new prototype.

7.3 Application Phases

Previous studies observe execution phases in SPEC CPU benchmark applications [62, 70]. To verify whether program phases exists in the evaluated cloud services, we collect time series of CPU metrics spanning 600 seconds, with sampling frequencies ranging from one second to ten seconds. We do not observe regular program phases with such sampling granularity. There may be program phases in the execution of individual requests, which range from tens of microseconds to tens of milliseconds. Nevertheless, they rarely manifest across requests, unless caused by load fluctuations, which Ditto captures. As the service processes hundreds of requests per second, the overall program phases are averaged across concurrent requests.

7.4 Multi-tenancy and Virtualization

We demonstrate Ditto's accuracy in local, virtualized, and multi-tenant deployments in Sections 6.2 and 6.5. Although we do not directly model contention and hypervisor events, the behavior of synthetic and target applications under multi-tenancy and virtualization is similar. In Ditto, the synthetic application clones the interference sensitivity by matching the target application's resource usage patterns. The performance overhead of virtualization comes primarily from network and disk I/O, nested paging, VM scheduling, and OS interactions [47]. Cloning the network and thread models, system calls and memory accesses ensures that synthetic and original applications experience similar overheads.

7.5 Limitations and Future Work

Ditto currently only supports C/C++ applications compiled for x86 ISA, as it relies on instrumentation tools designed for C/C++ and x86. However, Ditto does not depend on any language- or ISA-specific features, and can be easily adapted to other languages and ISAs.

Cloning for specialized hardware, such as GPUs, FPGAs, and smartNICs is also gaining in importance. Due to their radically different ISAs and computing paradigms, we leave extending Ditto for them to future work.

8 CONCLUSION

We presented Ditto, an accurate cloning framework for end-to-end monolithic services and microservices. Ditto captures the activity of an application across the system stack, including kernel and network events, and accurately reproduces its characteristics, decoupling representative cloud studies from access to production code.

ACKNOWLEDGMENTS

We sincerely thank Ramesh Illikkal, Yanqi Zhang, Nikita Lazarev, Zhuangzhuang Zhou, Daniel Sanchez, and the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was in part supported by an NSF CAREER Award CCF-1846046, NSF grant NeTS CSR-1704742, an Intel Research Award, an Intel Faculty Rising Star Award, a Sloan Research Fellowship, a Microsoft Research Fellowship, a Facebook Research Faculty Award, and a John and Norma Balen Sesquisentennial Faculty Fellowship.

REFERENCES

[1] Apache thrift. https://thrift.apache.org.
[2] giltene/wrk2. https://github.com/giltene/wrk2.
[3] grpc: A high performance open-source universal rpc framework. https://grpc.io/.
[4] Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/.
[5] Mongodb. https://www.mongodb.com.
[6] Nginx. https://nginx.org/en.
[7] Opentracing. https://opentracing.io/.
[8] Redis. https://redis.io.
[9] SPEC Cloud® IaaS 2018.
[10] stress-ng. https://wiki.ubuntu.com/Kernel/Reference/stress-ng.
[11] Zipkin. http://zipkin.io.
[12] perf: Linux profiling with performance counters, 2015.
[13] Mutated: A high-performance and very accurate load-generator for stressing servers and measuring their latency behaviour under load, 2016.
[14] tcpkali: A high performance tcp and websocket load generator and sink, 2017.
[15] 8 surprising facts about real docker adoption, 2018.
[16] iperf3: A tcp, udp, and sctp network bandwidth measurement tool, 2021.
[17] Andreas Abel and Jan Reineke. Uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 673–686, New York, NY, USA, 2019. Association for Computing Machinery.
[18] Andreas Abel and Jan Reineke. nanobench: A low-overhead tool for running microbenchmarks on x86 systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, August 2020.
[19] Dan Ardelean, Amer Diwan, and Chandra Erdman. Performance analysis of cloud applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 405–417, Renton, WA, April 2018. USENIX Association.
[20] Amro Awad and Yan Solihin. Stm: Cloning the spatial and temporal memory access behavior. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 237–247, 2014.
[21] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers. In *International Symposium on Computer Architecture (ISCA)*, 2019.
[22] Ganesh Balakrishnan and Yan Solihin. West: Cloning data cache behavior using stochastic traces. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, 2012.
[23] Luiz Barroso. Warehouse-scale computing: Entering the teenage decade. In *Proceedings of the 38th Intl. symposium on Computer architecture*, San Jose, CA, 2011.
[24] Luiz Barroso and Urs Hoelzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* MC Publishers, 2009.
[25] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, March 2017.
[26] Robert H Bell Jr and Lizy K John. Improved automatic testcase synthesis for performance model validation. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 111–120, 2005.
[27] Ramon Bertran, Alper Buyuktosunoglu, Meeta S. Gupta, Marc Gonzalez, and Pradip Bose. Systematic energy characterization of cmp/smt processor systems via automated micro-benchmarks. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 199–211, 2012.
[28] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site reliability engineering: How Google runs production systems.* " O'Reilly Media, Inc.", 2016.
[29] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, page 154–163, New York, NY, USA, 2006. Association for Computing Machinery.
[30] Philip Bille. A survey on tree edit distance and related problems. *Theoretical computer science*, 337(1-3):217–239, 2005.
[31] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011.
[32] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 41–42, New York, NY, USA, 2018. Association for Computing Machinery.
[33] Gerardo Canfora and Massimiliano Di Penta. New frontiers of reverse engineering. In *Future of Software Engineering (FOSE'07)*, pages 326–341. IEEE, 2007.
[34] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. Association for Computing Machinery.
[35] Mark Charney. X86 encoder decoder user guide, 2019.
[36] Shuang Chen, Christina Delimitrou, and Jose F. Martinez. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

[37] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. 2010.

[38] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, page 391–402, New York, NY, USA, 2008. Association for Computing Machinery.

[39] Deeksha Dangwal, Weilong Cui, Joseph McMahan, and Timothy Sherwood. Safer program behavior sharing through trace wringing. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1059–1072, 2019.

[40] Jeffrey Dean and Luiz Andre Barroso. The tail at scale. In *CACM, Vol. 56 No. 2*.

[41] Christina Delimitrou and Christos Kozyrakis. iBench: Quantifying Interference for Datacenter Workloads. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC)*. Portland, OR, September 2013.

[42] Christina Delimitrou and Christos Kozyrakis. Quality-of-Service-Aware Scheduling in Heterogeneous Datacenters with Paragon. In *IEEE Micro Special Issue on Top Picks from the Computer Architecture Conferences*. May/June 2014.

[43] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, UT, USA, 2014.

[44] eBPF Foundation. ebpf, 2021.

[45] Frank C. Eigler, Vara Prasad, Will Cohen, Hien Nguyen, Martin Hunt, Jim Keniston, and Brad Chen. Architecture of systemtap: a linux trace/probe tool, 2005.

[46] Qi Fan and Qingyang Wang. Performance comparison of web servers with different architectures: A case study using high concurrency workload. In *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pages 37–42. IEEE, 2015.

[47] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, 2015.

[48] Michael Ferdman, Almutaz Adileh, and et al. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proc. of ASPLOS*. London, England, UK, 2012.

[49] Brad Fitzpatrick. Distributed caching with memcached. In *Linux Journal, Volume 2004, Issue 124, 2004*.

[50] Agner Fog. The microarchitecture of intel, amd and via cpus an optimization guide for assembly programmers and compiler makers.

[51] Agner Fog et al. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. *Copenhagen University College of Engineering*, 93:110, 2011.

[52] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: Practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 135–151, New York, NY, USA, 2021. Association for Computing Machinery.

[53] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayantara Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

[54] Yu Gan, Yanqi Zhang, Kelvin Hu, Yuan He, Meghna Pancholi, Dailun Cheng, and Christina Delimitrou. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

[55] Karthik Ganesan, Jungho Jo, and Lizy K. John. Synthesizing memory-level parallelism aware miniature clones for SPEC CPU2006 and implantBench workloads. *ISPASS 2010 - IEEE International Symposium on Performance Analysis of Systems and Software*, pages 33–44, 2010.

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

[57] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[58] Tarun Goyal, Ajit Singh, and Aakanksha Agrawal. Cloudsim: simulator for cloud computing infrastructure and modeling. *Procedia Engineering*, 38:3566–3572, 2012.

[59] Brendan Gregg. *Systems performance: enterprise and the cloud*. Pearson Education, 2014.

[60] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2(11), 2011.

[61] 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 *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, 2001.

[62] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28, 2005.

[63] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1919–1928. PMLR, 10–15 Jul 2018.

[64] M. Haungs, P. Sallee, and M. Farrens. Branch transition rate: a new metric for improved branch classification analysis. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*, pages 241–250, 2000.

[65] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[66] Weizhe Hua, Zhiru Zhang, and G. Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[67] Intel. Intel vtune amplifier. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html, year = 2021,.

[68] Intel. Intel software development emulator, 2012.

[69] Intel. Dynamic control-flow graph generation with pinplay, 2015.

[70] Canturk Isci, Alper Buyuktosunoglu, and Margaret Martonosi. Long-term workload phases: Duration predictions and applications to dvfs. *Ieee Micro*, 25(5):39–51, 2005.

[71] Tara Merin John, Syed Kamran Haider, Hamza Omar, and Marten van Dijk. Connecting the dots: Privacy leakage via write-access patterns to the main memory. *IEEE Transactions on Dependable and Secure Computing*, 17(2):436–442, 2020.

[72] Ajay Joshi, Lieven Eeckhout, Robert H. Bell, and Lizy John. Performance cloning: A technique for disseminating proprietary applications as benchmarks. In *2006 IEEE International Symposium on Workload Characterization*, pages 105–115, 2006.

[73] Ajay Joshi, Lieven Eeckhout, Robert H. Bell, and Lizy K. John. Distilling the essence of proprietary workloads into miniature benchmarks. *ACM Trans. Archit. Code Optim.*, 5(2), September 2008.

[74] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News*, 43(3S):158–169, June 2015.

[75] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer architecture letters*, 15(1):45–49, 2015.

[76] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 36–51, New York, NY, USA, 2021. Association for Computing Machinery.

[77] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proc. of EuroSys*. 2014.

[78] Chit-Kwan Lin and Stephen J. Tarsa. Branch prediction is not a solved problem: Measurements, opportunities, and future directions. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 228–238, 2019.

[79] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proc. of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*. Portland, OR, 2015.

[80] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 412–426, New York, NY, USA, 2021. Association for Computing Machinery.

[81] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 319–330, 2011.

[82] David Meisner, Junjie Wu, and Thomas F. Wenisch. Bighouse: A simulation infrastructure for data center systems. In *2012 IEEE International Symposium on Performance Analysis of Systems Software*, pages 35–45, 2012.

[83] Fionn Murtagh and Pedro Contreras. Algorithms for hierarchical clustering: an overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(1):86–97, 2012.

[84] Intel® 64 and ia-32 architectures optimization reference manual. February 2022.

[85] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[86] Reena Panda and Lizy K. John. Halo: A hierarchical memory access locality modeling technique for memory system explorations. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, page 118–128, New York, NY, USA, 2018. Association for Computing Machinery.

[87] Reena Panda and Lizy Kurian John. Proxy benchmarks for emerging big-data workloads. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 105–116, 2017.

[88] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, Amol Shukla, and David R Cheriton. Comparing the performance of web server architectures. *ACM SIGOPS Operating Systems Review*, 41(3):231–243, 2007.

[89] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, page 2–11, New York, NY, USA, 2010. Association for Computing Machinery.

[90] G. Ravi, R. Bertran, P. Bose, and M. Lipasti. Micrograd: A centralized framework for workload cloning and stress testing. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 70–72, Los Alamitos, CA, USA, mar 2021. IEEE Computer Society.

[91] Charles Reiss, Alexey Tumanov, Gregory Ganger, Randy Katz, and Michael Kozych. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of SOCC*. 2012.

[92] Michiel Ronsse and Koen De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, may 1999.

[93] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[94] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 475–486, New York, NY, USA, 2013. Association for Computing Machinery.

[95] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 488–498, New York, NY, USA, 2013. Association for Computing Machinery.

[96] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[97] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, Armando Fox, and David Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proc. of CCA*, volume 8, page 228, 2008.

[98] A. Sriraman and T. F. Wenisch. μ suite: A benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12, 2018.

[99] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 733–750, New York, NY, USA, 2020. Association for Computing Machinery.

[100] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F. Wenisch. Softsku: Optimizing server architectures for microservice diversity @scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 513–526, New York, NY, USA, 2019. Association for Computing Machinery.

[101] Akshitha Sriraman and Thomas F. Wenisch. μtune: Auto-tuned threading for OLDI microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, Carlsbad, CA, October 2018. USENIX Association.

[102] W Richard Stevens and Thomas Narten. Unix network programming. *ACM SIGCOMM Computer Communication Review*, 20(2):8–9, 1990.

[103] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. Workload characterization for microservices. In *Proc. of IISWC*. 2016.

[104] Luk Van Ertvelde and Lieven Eeckhout. Dispersing proprietary applications as benchmarks through code mutation. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 201–210, 2008.

[105] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. Cachequery: Learning replacement policies from hardware caches. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 519–532, New York, NY, USA, 2020. Association for Computing Machinery.

[106] Qingyang Wang, Chien-An Lai, Yasuhiko Kanemasa, Shungeng Zhang, and Calton Pu. A study of long-tail latency in n-tier systems: Rpc vs. asynchronous invocations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 207–217. IEEE, 2017.

[107] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, 2014.

[108] Shungeng Zhang, Qingyang Wang, Yasuhiko Kanemasa, Huasong Shan, and Liting Hu. The impact of event processing flow on asynchronous server efficiency. *IEEE Transactions on Parallel and Distributed Systems*, 31(3):565–579, 2019.

[109] Yanqi Zhang, Yu Gan, and Christina Delimitrou. μqsim: Enabling accurate and scalable simulation for interactive microservices. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 212–222, 2019.

[110] Yanqi Zhang, Iñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, October 2021.

[111] Yuhao Zhu, Daniel Richins, Matthew Halpern, and Vijay Janapa Reddi. Microarchitectural implications of event-driven server-side web applications. In *Proc. of MICRO*, 2015.

Received 2022-07-07; accepted 2022-09-22