End-to-End Cloud Application Cloning With Ditto

Mingyu Liang, *Cornell University, Ithaca, NY, 14853, USA*

Yu Gan, *Google, Mountain View, CA, 94043, USA*

Yueying Li, *Cornell University, Ithaca, NY, 14853, USA*

Carlos Torres and Abhishek Dhanotia, *Meta, Cambridge, MA, 02140, USA*

Mahesh Ketkar, *Intel, Portland, OR, 97229, USA*

Christina Delimitrou, *Massachusetts Institute of Technology, Cambridge, MA, 02139, USA*

The lack of publicly available cloud services has been a recurring problem in architecture and systems. Although open source benchmarks exist, they do not capture the complexity of cloud services. Application cloning is a promising approach, however, prior work is limited to CPU-/cache-centric, single-node services. We present Ditto, a framework for cloning end-to-end cloud applications and monolithic and microservices that captures input–output and network activity as well as kernel operations, in addition to application logic. Ditto takes a hierarchical approach to application cloning, capturing the dependency graph across services, recreating each tier's control/dataflow, and generating system calls and assembly that mimics individual applications. Ditto does not reveal the logic of the original application, facilitating publicly sharing clones of production services. We show that across a diverse set of applications, Ditto accurately captures their resource characteristics as well as their performance metrics, is portable across platforms, and facilitates a wide range of studies.

Cloud computing now hosts a large fraction of the world's computations, ranging from machine learning workloads to latency-critical interactive services.[1] Understanding these applications is imperative to correctly design the systems that populate future cloud infrastructures.

Directly executing real-world applications using representative workloads provides the most accurate insights into their behavior. However, due to factors such as privacy and intellectual property (IP) concerns, such applications and workloads often remain inaccessible to researchers. To address this challenge, alternative methodologies have been proposed. These can be broadly classified into three categories: open source benchmarks,[4] simulation and trace replay,[8] and application performance cloning with synthetic benchmarks.[6]

Each of these approaches presents inherent trade-offs. Open source benchmarks, although flexible, often fail to mirror the complexity and evolving nature of production cloud deployments. Simulation and trace replay provide greater realism, but lack flexibility; the results are constrained by the original system configuration under which the trace was captured. Synthetic benchmarks seek a balance by modeling key aspects of the target application while retaining adaptability. However, existing techniques for synthetic benchmark cloning focus largely on CPU-centric, single-tier, user-level applications.[6]

When evaluating cloud workloads, focusing solely on CPU-centric microarchitectural events provides an incomplete picture. Cloud services inherently dedicate significant resources to networking and operating

0272-1732 © 2024 IEEE
Digital Object Identifier 10.1109/MM.2024.3419067
Date of publication 1 July 2024; date of current version 14 August 2024.

system (OS)-level operations. Moreover, their distributed nature, composed of interdependent components, demands that cloning efforts capture this complex, multitier behavior. Additionally, an assembly-level focus on metrics like instructions per cycle (IPC), cache miss rate, and dependency distance neglects the crucial higher-level performance indicators that cloud services prioritize, such as average and tail latency.

Our article on *Ditto*,[9] presented at the 2023 ACM International Conference on Architectural Support for Programming Languages and Operating Systems, addresses these limitations. Ditto is an application cloning framework designed for the cloud era. It automatically reproduces an end-to-end application structure as well as key performance characteristics of distributed services, from monolithic applications to complex microservice topologies. In particular, Ditto transcends traditional cloning limitations by mirroring behavior across the entire system stack: hardware, input–output (I/O), networking, and OS. More importantly, Ditto does not reveal any code or high-level functionality of the original application, which motivates researchers to share and study realistic cloud application clones without compromising sensitive data or IP.

Ditto employs several key techniques for transparently cloning application topologies. First, it leverages distributed tracing tools to capture cross-service dependency graphs. Second, it reconstructs internal service control and dataflow using thread and network I/O modeling. Finally, Ditto generates appropriate system calls and user-space assembly code to replicate both on-CPU and off-CPU behavior. The application cloning process is entirely automated. Ditto's methodology can be adapted to different platforms, deployments, and application configurations, such as load and thread pools, without requiring retraining, ensuring that synthetic applications closely mirror their production counterparts.

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 specifications to hardware vendors using the synthetic workloads, and researchers can use representative end-to-end cloud services without the need for production code access. Ditto is open source software.[a]

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

APPLICATION CLONING ACROSS THE SYSTEM STACK

Application cloning for cloud services is challenging due to the complexity and heterogeneity of the implementation and the various platforms on which they can be deployed. Different services can have entirely different bottlenecks across different systems stacks, ranging from hardware to the application layer. For example, key-value stores (KVS) require high single-core performance and memory bandwidth to retrieve a large number of data under a strict latency service-level objective, while databases are usually bottlenecked by disk I/O bandwidth. 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 demonstrates an abstract view of a generic system stack for a single cloud server.[5] The performance of an application is determined by factors that range from the application code and inputs to the environment that it is running on, including containerization technology, the hypervisor, server platforms, and any co-located applications. In the next section, we briefly describe why these factors matter.

IN PARTICULAR, DITTO TRANSCENDS TRADITIONAL CLONING LIMITATIONS BY MIRRORING BEHAVIOR ACROSS THE ENTIRE SYSTEM STACK: HARDWARE, INPUT–OUTPUT, NETWORKING, AND OS.

Application Inputs

The behavior and performance of cloud applications are significantly impacted by the service configuration and input load, with the latter going through well-documented fluctuations.[2] The application's configuration, although it changes 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.

Application Codebase and Binary

The application and its linked libraries are intrinsic to its performance, regardless of the platform on which it is deployed. Modifications in the application code can alter the control and dataflow of a service, its memory access patterns, and its resource bottlenecks. This is

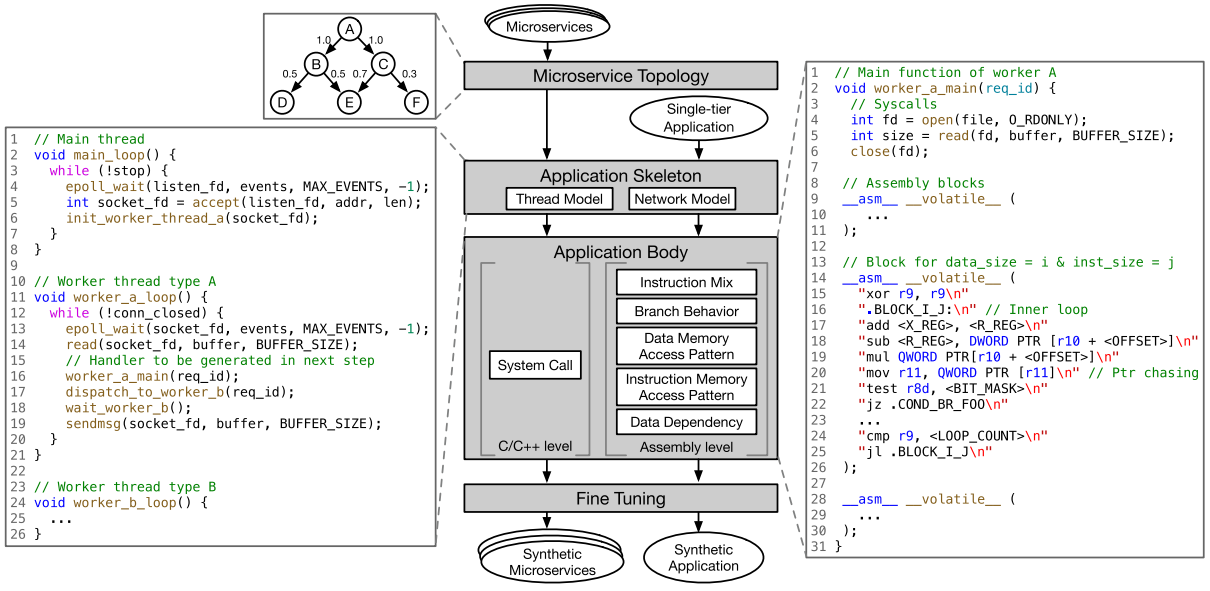

FIGURE 1. General system stack for cloud applications.[5] The dashed boxes are optional layers for virtualization.

especially true for new cloud programming frameworks, like microservices and serverless, where services are updated on a daily basis.

Deployment Environment

Containers and Virtual Machines

Cloud services are often deployed with containers and/or virtual machines. These add different levels of performance overheads, primarily due to the extra I/Os and network layers.[3] 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.

OS Kernel

Cloud applications are especially dependent on OS performance, given that they spend a large fraction of their execution at the kernel level for interrupt handling, I/O requests, memory management, task scheduling, and so on.[4] Prior work on application cloning has focused mostly on user-level application logic; cloud services that overlook kernel operations leads to very different performance characteristics compared to the original application.

CPU Memory Subsystem

The CPU's memory subsystem is a dominant factor in cloud application performance, even for services that spend significant time processing network requests. We follow the top-down analysis methodology in Yasin[12] to identify the key CPU performance metrics that impact the overall IPC, and reproduce them in synthetic applications.

Hardware Devices

Services interact with hardware devices, including disks, and network interface cards 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.

Multitenancy

Multitenancy improves data center utilization by deploying multiple services on the same node. Applications share resources, including CPU cores, last level cache (LLC), and memory, disk I/O, and network bandwidth.[10] Resource contention can degrade performance and should be accounted for in the application cloning process.

SYSTEM DESIGN AND IMPLEMENTATION

Ditto is an application cloning framework for both single-tier and microservice applications. It generates services that faithfully reproduce the performance, resource profile, and thread-level control/dataflow of the original workload, decoupling representative

system studies from access to the source code or binary of production cloud services.

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, remote procedure call (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, exposing only the skeleton and postprocessed 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.

Figure 2 shows an overview of Ditto's profiling and generation process. If the target service consists of a set of microservices, Ditto first learns their RPC dependency graph, using distributed tracing. This graph is then used to generate the application programming interfaces among the different synthetic microservices. Next, Ditto analyzes the thread and networking model, e.g., single- or multithreaded, and synchronous or asynchronous, respectively, using kernel-level profiling, and builds the skeleton of each service. The application skeleton contains empty handlers that 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

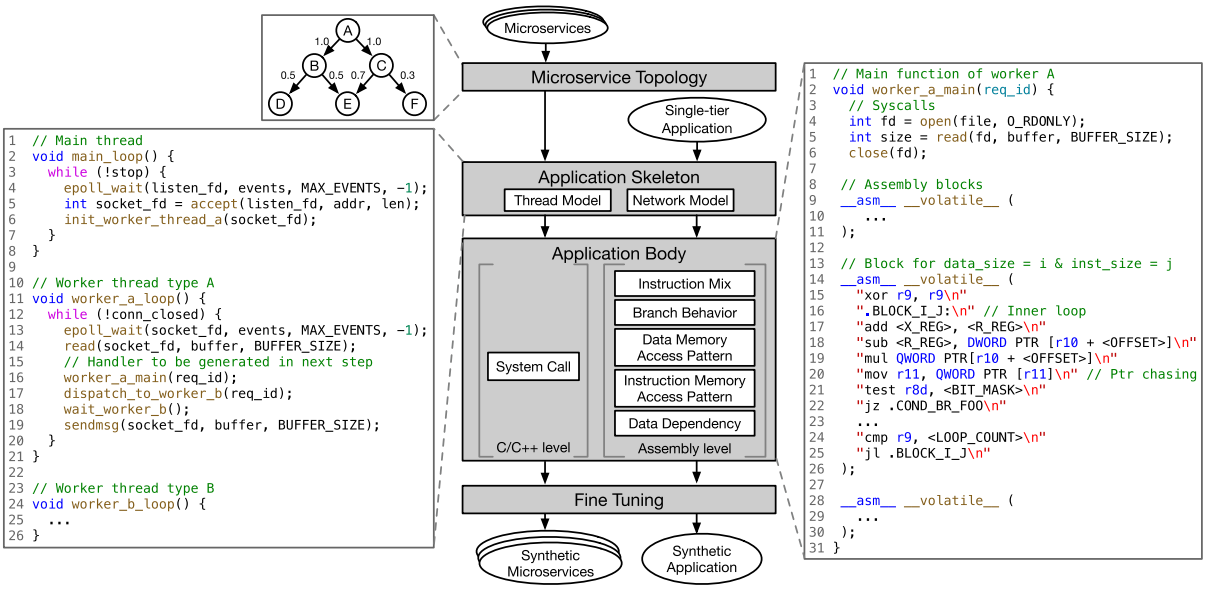

FIGURE 2. Overview of Ditto's synthetic benchmark-generation process.

as a performance and resource proxy for the original service.

Microservice Topology

A topology of microservices is a directed acyclic graph, where the nodes are microservices and the edges indicate the dataflow between dependent tiers. Ditto leverages the distributed tracing 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. It then automatically extracts the dependency graph between microservices and uses it as input to the skeleton generator.

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 that face tight latency constraints as it directly impacts their performance and scalability.

The network model defines the mechanisms through which an application interacts with other services. An application can operate as a client, server, or a combination of the two. Client-side services often employ either synchronous or asynchronous communication paradigms. On the server side, network models typically include blocking, nonblocking, and I/O multiplexing. Ditto uses SystemTap to profile the network model by probing kernel-space functions and data structures. It then chooses one out of several network models that combine the different design choices described earlier to match the profiled network configurations.

Cloud services frequently leverage multithreading to facilitate asynchronous I/O operations and enable parallel processing. To characterize these threading patterns, Ditto utilizes SystemTap to conduct a call stack analysis. This provides insights into the functionality, lifecycles, and invocation points of threads within a target application. Threads are subsequently clustered based on these shared characteristics. During the generation of synthetic counterparts, Ditto emulates the observed threading behavior via a set of threads that is dedicated to executing synthetic code that is designed to mirror the profiled application's behavior.

Application Body

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

To replicate kernel-space performance, Ditto profiles system calls, excluding those that are focused on network handling and process management that are explicitly modeled in the previous step. Using SystemTap, Ditto captures the distribution of these system calls, including counts and arguments, to precisely characterize kernel-level behavior. These data inform the generation of synthetic applications that mirror the original system's kernel-space patterns.

At the user level, Ditto analyzes the factors that significantly impact application on-CPU performance. These factors include instruction mix, memory access patterns (data and instruction), branch behavior, and data dependencies.[11,12] To collect platform-independent metrics, Ditto employs a suite of tools including SystemTap, Intel Software Development Emulator (SDE), and Valgrind. These data guide the generation of synthetic applications using carefully crafted inline assembly code, which exhibits similar user-level on-CPU performance characteristics without disclosing the original code's functionality. The use of platform-independent metrics ensures that Ditto-generated synthetic applications can be ported to other platforms without the need for additional profiling.

THE APPLICATION SKELETON IS A CRITICAL DESIGN CHOICE FOR CLOUD SERVICES THAT FACE TIGHT LATENCY CONSTRAINTS AS IT DIRECTLY IMPACTS THEIR PERFORMANCE AND SCALABILITY.

Fine-Tuning

Finally, Ditto implements fine-tuning to counterbalance the impact of the instrumentation tools themselves. Ditto iteratively runs the synthetic application, computes the errors between target and synthetic services, 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. As relationships between knobs and performance are locally linear, we use a feedback-based heuristic to tune the knobs.

Implementation

Ditto is implemented primarily in Python and C in approximately 16,000 lines of code. It supports C/C++ applications, the Apache Thrift and gRPC RPC frameworks, and x86 instruction set architectures (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 distributed in a server cluster, using Docker Swarm or Kubernetes. Although the runtime profilers and emulators, including SystemTap, Intel SDE, and Valgrind, can introduce overheads to the original application during profiling, this overhead occurs only once and does not affect the accuracy of the platform-independent features collected during profiling.

To generate a clone, cloud providers need only 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 the 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. We have been able to run binaries synthesized by Ditto directly on hardware as well as on execution-driven simulators, including gem5 and ZSim, and trace-driven simulators like Ramulator.

EVALUATION

Methodology

We evaluate Ditto across a diverse set of services, including KVSs (Memcached and Redis), web servers (NGINX), databases (MongoDB), and complex microservices (Social Network). To generate input loads for different services, we employ tools like wrk2, YCSB, tcpkali, and an open-loop variant of Mutated. For all the synthetic applications, we use the same load generator as the original application.

Ditto is validated on a heterogeneous cluster, with two types of servers. All the servers run x86 ISA, but differ in the CPU and memory architectures, and their storage and network.

Validation

Validation on Varying Loads

Figure 3 shows the CPU, network, and disk performance metrics, and latency for six applications under

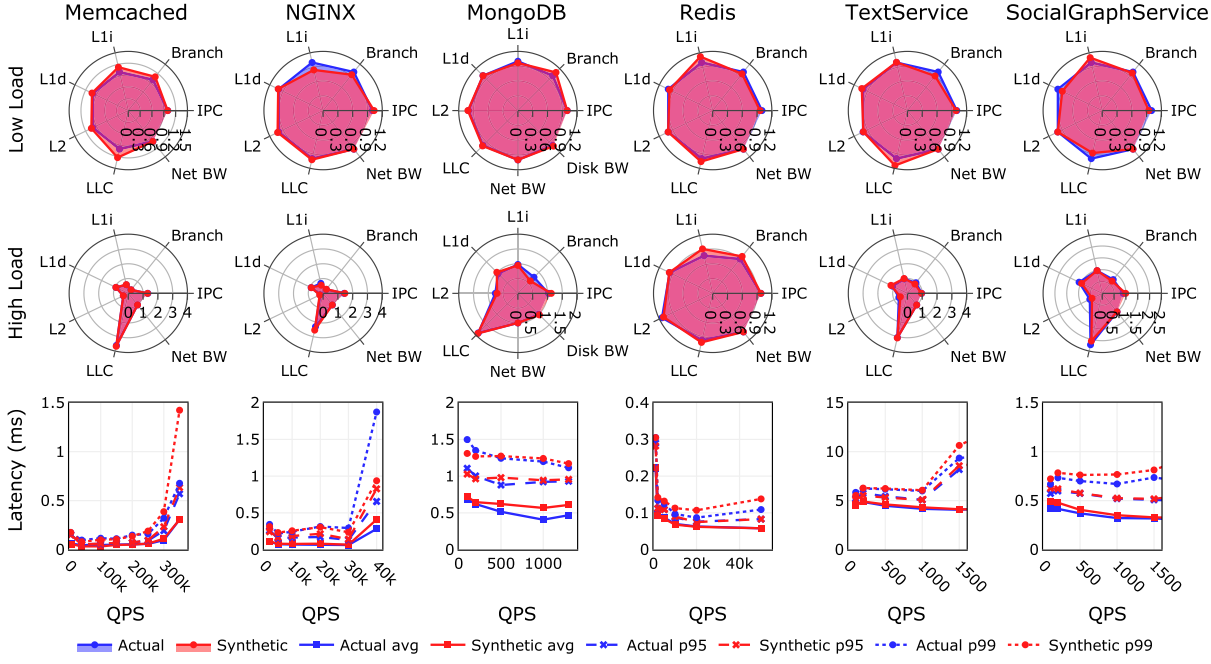

FIGURE 3. CPU performance metrics (IPC, branch mispredictions, L1i, L1d, L2, and LLC miss rates), network bandwidth (net BW), disk BW (MongoDB only), and service latency under varying load across six services. CPU metrics are normalized to each original application's metrics under low load. Net BW and disk BW are, by exception, normalized to each original application's BW under current load because their magnitudes change significantly, and would obscure the figure's shape. avg: average.

different QPS on platform A. In addition to the four single-tier applications, we also show the resource characteristics for TextService and SocialGraphService, two applications in Social Network, which are representative of the other tiers of the service. All the applications are generated using profiling data under a single load configuration; *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 a CPU for the other applications).

The top two rows show IPC; branch misprediction; L1i (level 1i), L1d, L2, and LLC miss rates; and network and disk I/O bandwidth under low and high load, with the average errors across all the applications being 4.1%, 9.9%, 7.1%, 5.1%, 6.9%, 12.1%, 0.1%, and 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, Redis maintains metrics similar to those observed under low load. In contrast, the five other applications demonstrate varying degrees of change in L2 misses, LLC misses, and branch mispredictions. 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 bandwidths also conform to the original ones by faithfully reproducing the system calls. We show only the disk bandwidth for MongoDB as 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. As 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. Although the end-to-end latency of Social Network increases at high load, the latency of TextService and SocialGraphService increases only slightly as they are not bottleneck tiers.

Figure 4 shows the end-to-end latency of the original and the synthetic Social Network when every individual microservice is replaced with a synthetic one. Both the end-to-end latency and saturation point closely match across loads.

Validation on Varying Platforms

We validate the CPU, network, and disk performance metrics, along with service latencies across different x86 platforms. Each application is initially profiled on platform A, with subsequent validations conducted on platforms A and B (Table 1). Figure 5 shows that the synthetic benchmarks react to platform changes in a way similar to that of the original applications. More specifically, all six applications have different degrees of L2 cache miss increases on platforms B due to their smaller L2 cache sizes. The applications running on platform B, which is an older CPU generation, have consistently lower IPC. The network and disk I/O bandwidths are identical across platforms because the numbers of data transferred are independent of the platform.

The bottom line plots how the latency on the two platforms, where the synthetic always matches the original. All the 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

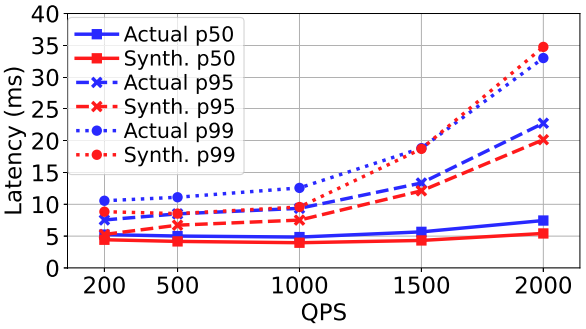

FIGURE 4. End-to-end latency for the social network. synth: synthetic.

TABLE 1. Server platform specifications.

	Platform A	Platform B
CPU model	Gold 6152	E5-2660 v3
Base frequency	2.1 GHz	2.6 GHz
CPU cores	22	10
CPU family	Skylake	Haswell
Sockets	2	2
L1i/L1d	32 KB/32 KB	32 KB/32 KB
L2	1 MB	256 KB
LLC	30.25 MB	25 MB
RAM	192 GB at 2666	128 GB at 2400
Disk	1-TB SSD	2-TB HDD
Network	10 Gbe	1 Gbe

RAM: random-access memory; SSD: solid-state drive; HDD: hard disk drive.

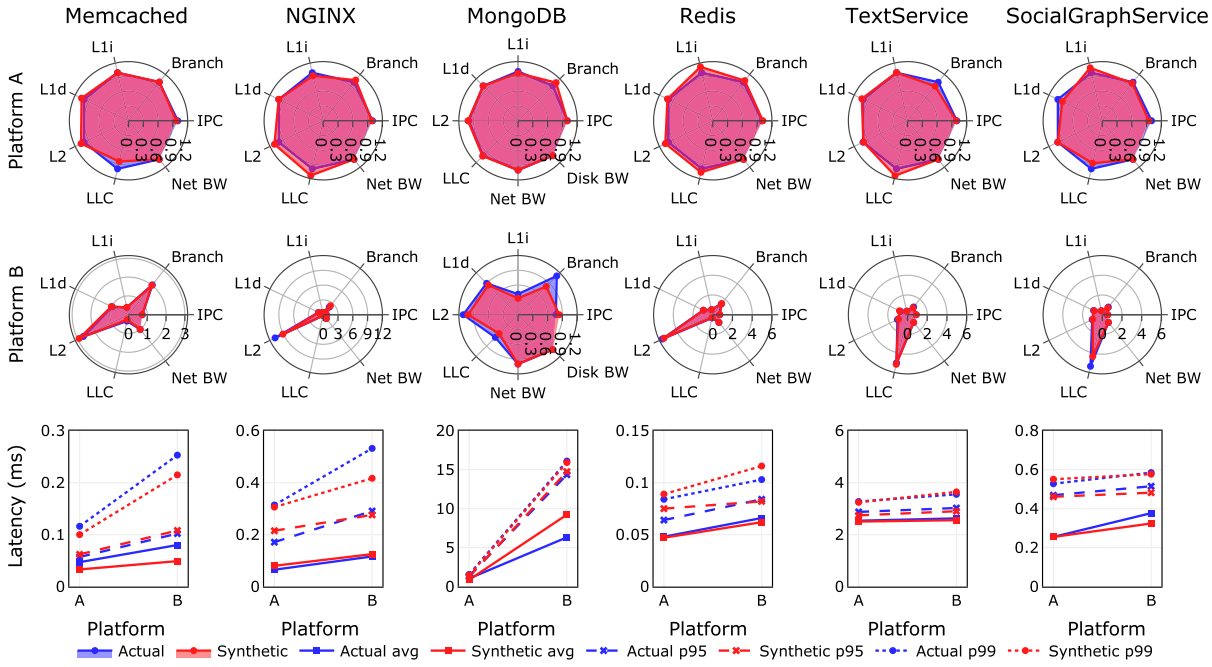

FIGURE 5. CPU metrics (IPC, branch misprediction, L1i, L1d, L2, and LLC misses), net BW, disk BW (MongoDB only) and latencies across platforms. CPU metrics are normalized to each original service on Platform A.

solid-state drives. In general, because 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.

Case Study: CPU Core and Frequency Scaling

Figure 6 shows the use of Ditto to evaluate power management in Memcached with CPU core and frequency scaling. Each cell represents p99 latency under a given number of cores and frequency. We set the quality of service (QoS) as 1 ms, and the cells with marks mean that the QoS cannot be satisfied for that configuration. Memcached cannot meet the QoS at low frequency even with the maximum number of cores, which prohibits its 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.

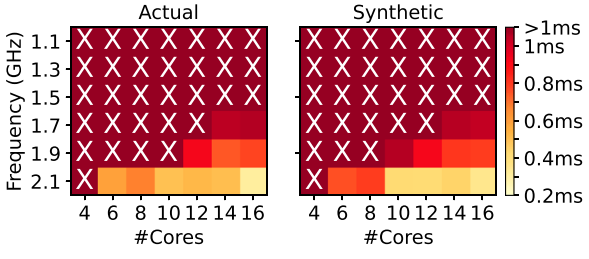

FIGURE 6. Ninety-ninth percentile latency of actual and synthetic Memcached under varying CPU frequency core count.

DISCUSSION

End-to-End Application Cloning

Ditto is the first framework to facilitate end-to-end cloning of distributed cloud applications. Although cloning has proven effective for microarchitectural analysis, its broader impact on cloud systems has been constrained due to the limitations of prior work when trying to capture the application's activity across the full system stack.

This is essential for cloud services and has required a carefully engineered tiered approach to ensure that capturing this information does not result in an impractical, hard-to-use framework. Additionally, Ditto is designed to be modular, with each layer of the framework being easily replaceable to increase or decrease the detail of cloning. For example, a user can easily replace the default memory access generation layer with a memory access replay tool, which would be more appropriate for data-locality studies.

Simplifying the "What-If" Simulation

Estimating how a change to an application's design or deployment will impact both its performance and resource needs is both critical and challenging. Critical, because it allows application developers and cloud operators to evaluate whether that change will be beneficial in the long term, and challenging, because without direct, expensive experimentation, estimating the impact of a change is very difficult, especially for complex, multitier application topologies.

Frameworks like Ditto offer a powerful solution. Consider a microservice provider who is evaluating a shift from RPC-based communication to message passing. With Ditto, the provider can seamlessly swap one communication framework for the other for rapid assessment. This contrasts starkly with the significant implementation effort that would be required to deploy this change in a production environment.

Similarly, Ditto can easily assess the impact that changing the application's instruction mix, data, and instruction footprint, or the number of data transfers over the network would have on performance and resource usage. This requires simply adjusting a few configuration knobs in the cloning framework, and without actually making these changes in the original application.

Finally, Ditto empowers developers to investigate appropriate service granularities for their applications. By manipulating the communication-to-computation ratio for service tiers, developers can evaluate the performance implications of making their tiers more or less fine-grained, without the substantial overhead of redesigning each application version. This is invaluable, given the profound effect service granularity has on end-to-end performance.[4,7]

Enabling Realistic Cloud Studies Without Access to Production Code

The scarcity of realistic cloud application benchmarks presents a persistent challenge within the architecture and system communities. Although open source benchmarks offer value, they fall short of replicating the intricate dynamics and scale of production-level services.

This lack of publicly available benchmarks extends beyond academia, profoundly impacting industry practices. When cloud providers seek to acquire next-generation servers, they are unable to share production services with hardware vendors for benchmarking purposes due to IP concerns. Consequently, they often rely on legacy benchmarks like SPECCPU and SPECJBB, which poorly reflect the nature of contemporary cloud applications. Ditto addresses this limitation by enabling cloud providers to generate end-to-end proxies of their services. These proxies allow secure sharing with hardware vendors without compromising IP.

In fact, in the short time since its publication, Ditto has already been extensively used by hardware vendors and cloud providers as an application cloning framework for benchmarking next-generation servers.

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 supported in part by a U.S. National Science Foundation CAREER Award CCF-1846046, an Intel Research Award, Intel Faculty Rising Star Award, a Sloan Research Fellowship, Microsoft Research Fellowship, and a Facebook Research Faculty Award.

REFERENCES

1. L. Barroso and U. Hoelzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. San Rafael, CA: MC Publishers, 2009.
2. C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Operating Syst. (ASPLOS)*, Salt Lake City, UT, USA, 2014, pp. 127–144, doi: 10.1145/2644865.2541941.
3. W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *Proc. 2015 IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Philadelphia, PA, USA, 2015, pp. 171–172, doi: 10.1109/ISPASS.2015.7095802.
4. Y. Gan et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Operating Syst. (ASPLOS)*, Apr. 2019, pp. 3–18, doi: 10.1145/3297858.3304013.
5. B. Gregg, *Systems Performance: Enterprise and the Cloud*. London, U.K.: Pearson Education, 2014.
6. A. Joshi, L. Eeckhout, R. H. Bell, and L. John, "Performance cloning: A technique for disseminating proprietary applications as benchmarks," in *Proc. IEEE Int. Symp. Workload Characterization*, 2006, pp. 105–115, doi: 10.1109/IISWC.2006.302734.
7. N. Lazarev, N. Adit, S. Xiang, Z. Zhang, and C. Delimitrou, "Dagger: Towards efficient RPCs in cloud microservices with near-memory reconfigurable

NICs," *IEEE Comput. Archit. Lett.*, vol. 19, no. 2, pp. 134–138, Jul./Dec. 2020, doi: 10.1109/LCA.2020.3020064.
8. M. Liang et al., "Mystique: Enabling accurate and scalable generation of production AI benchmarks," in *Proc. 50th Annu. Int. Symp. Comput. Archit.*, 2023, pp. 1–13, doi: 10.1145/3579371.3589072.
9. M. Liang et al., "Ditto: End-to-end application cloning for networked cloud services," in *Proc. 28th ACM Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2023, vol. 2, pp. 222–236, doi: 10.1145/3575693.3575751.
10. D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, Portland, OR, USA, 2015, pp. 450–462.
11. R. Panda and L. K. John, "Proxy benchmarks for emerging big-data workloads," in *Proc. 26th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, 2017, pp. 105–116, doi: 10.1109/PACT.2017.44.
12. A. Yasin, "A top-down method for performance analysis and counters architecture," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2014, pp. 35–44, doi: 10.1109/ISPASS.2014.6844459.

MINGYU LIANG is a Ph.D. candidate in computer engineering at the School of Electrical and Computer Engineering, Cornell University, Ithaca, NY, 14853, USA. His research interests include computer architecture, cloud computing, and new cloud programming models. Liang received his master's degree in electrical and computer engineering from Cornell University. He is a Student Member of IEEE and the Association for Computing Machinery. Contact him at ml2585@cornell.edu.

YU GAN is research engineer in computer science at Google, Mountain View, CA, 94043, USA. His research interests include the intersection of distributed systems and machine learning. Gan received his Ph.D. degree in electrical and computer engineering from Cornell University. He is a member of the Association for Computing Machinery. Contact him at yg397@cornell.edu.

YUEYING LI is a Ph.D. candidate in computer science at the School of Computer Science, Cornell University, Ithaca, NY, 14853, USA. Her research interests include machine learning systems, data center computing, and computer architecture. Li received her master's degree in computer science from Cornell University. She is a Student Member of IEEE and the Association for Computing Machinery. Contact her at yl3469@cornell.edu.

CARLOS TORRES is a staff performance engineer with Meta's Co-Design team, Cambridge, MA, 02140, USA. His research interests include workload characterization and hardware/software optimizations for Meta's large-scale distributed services. Torres received his bachelor's degree in computer engineering from the University of Puerto Rico. Contact him at cltorres@meta.com.

ABHISHEK DHANOTIA is a technical leader at Meta, Cambridge, MA, 02140, USA. His research interests include system architecture and performance and efficiency of Meta's next-generation compute platforms. Dhanotia received his master's degree in computer engineering from the North Carolina State University. Contact him at abhishekd@meta.com.

MAHESH KETKAR is a principal engineer and manager at Intel, Portland, OR, 97229, USA. His research interests include the development of performance analysis and optimization technologies that span the microarchitecture to system levels. Ketkar received his Ph.D. degree in electrical engineering from the University of Minnesota. Contact him at mahesh.c.ketkar@intel.com.

CHRISTINA DELIMITROU is an associate professor with the Electrical Engineering and Computer Science Department, Massachusetts Institute of Technology, Cambridge, MA, 02139, USA. Her research interests include computer architecture and distributed systems. Delimitrou received her Ph.D. degree in electrical engineering from Stanford University. She is a Member of IEEE and the Association for Computing Machinery. Contact her at delimitrou@csail.mit.edu.