This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3419067

End-to-End Cloud Application Cloning with Ditto

Mingyu Liang
Cornell University
ml2585@cornell.edu

Yu Gan
Cornell University
yg397@cornell.edu

Yueying Li
Cornell University
yl3469@cornell.edu

Carlos Torres
Meta
cltorres@meta.com

Abhishek Dhanotia
Meta
abhishekd@meta.com

Mahesh Ketkar
Intel
mahesh.c.ketkar@intel.com

Christina Delimitrou
MIT
delimitrou@csail.mit.edu

***Abstract*—The lack of publicly-available cloud services has been a recurring problem in architecture and systems. While 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, 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, capturing the dependency graph across services, recreating each tier's control/data flow, 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.

I. Introduction

Cloud computing now hosts a large fraction of the world's computation, 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 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, while flexible, often fail to mirror the complexity and evolving nature of production cloud deployments. Simulation and trace replay provide greater realism, but lack flexibility; 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 largely focus 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 OS-level operations. Moreover, their distributed nature, composed of interdependent components, demands that cloning efforts capture this complex, multi-tier behavior. Additionally, an assembly-level focus on metrics like 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 paper on *Ditto* [9], presented in ASPLOS'23, addresses these limitations. Ditto is an application cloning framework designed for the cloud era. It automatically reproduces the 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, 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 intellectual property.

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 data flow 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. The methodology of Ditto can be adapted to different platforms, deployments and application configurations, such as load and thread pools without requiring retraining, ensuring 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 specs to hardware vendors using the synthetic workloads, and researchers can use representative end-to-end cloud services without the need

© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3419067

for production code access. Ditto is open-source software. [1]

II. 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 they can be deployed on. Different services can have entirely different bottlenecks across different systems stacks ranging from hardware to application layer. For example, key-value stores (KVS) require high single-core performance and memory bandwidth to retrieve a large amount of data under a strict latency SLO, 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.

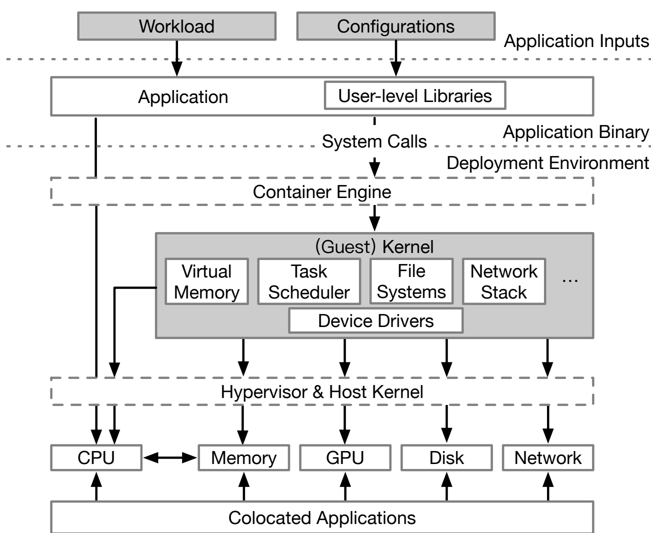

Fig. 1: General system stack for cloud applications [5]. Dashed boxes are optional layers for virtualization.

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 it is running on, including containerization technology, the hypervisor, server platforms, and any collocated applications. We briefly describe why these factors matter below.

A. 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 [2]. 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.

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

B. 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.

C. Deployment Environment

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 [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.

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. [4]. 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) 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. We follow the top-down analysis methodology in [12] to identify the key CPU performance metrics that impact the overall IPC and reproduce them in the synthetic applications.

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.

D. 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 [10]. Resource contention can degrade performance, and should be accounted for in the application cloning process.

III. 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/data flow of the original workload, decoupling representative system studies from access to the source code or binary of production cloud services.

© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3419067

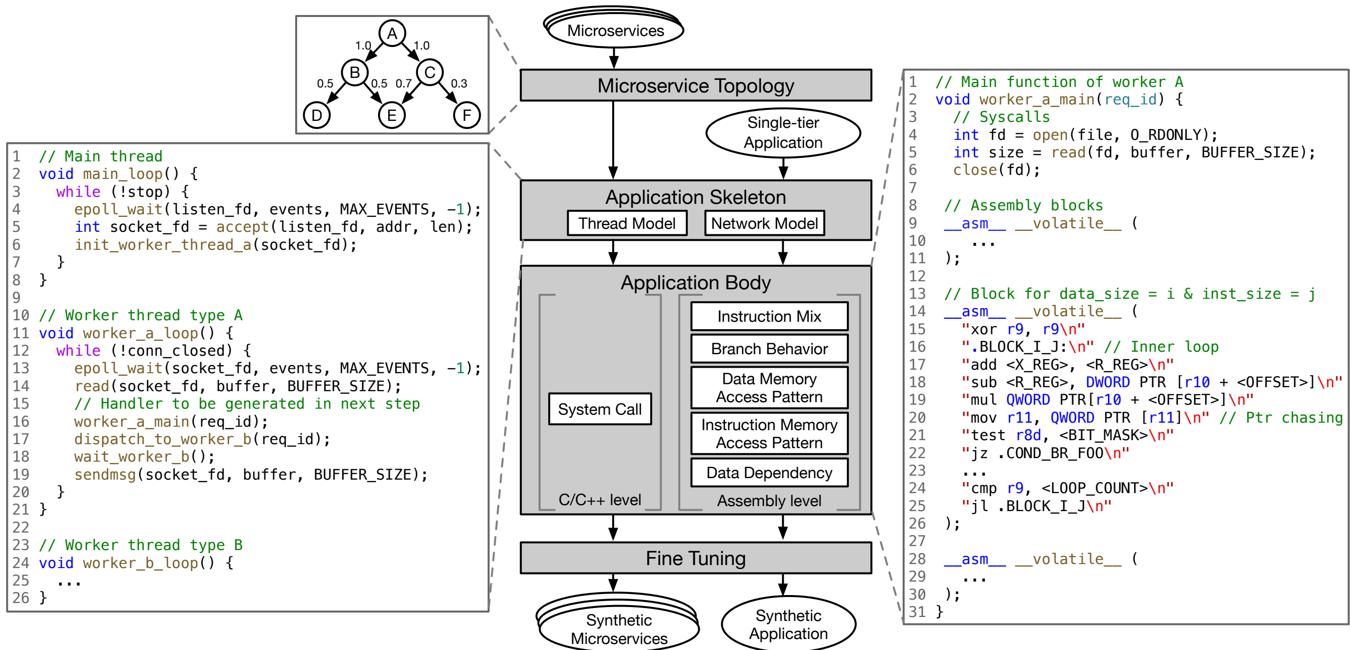

Fig. 2: Overview of Ditto's synthetic benchmark generation process.

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.

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 Remote Procedure Call (RPC) dependency graph, using distributed tracing. 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.

A. *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. 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.

B. *Application Skeleton*

We define the application skeleton as the network and thread models of an application, which determine how it handles

© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3419067

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, 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, a 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, non-blocking, 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 above, 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 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 dedicated to executing synthetic code designed to mirror the profiled application's behavior.

C. 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.

To replicate kernel-space performance, Ditto profiles system calls excluding those 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. This data informs the generation of synthetic applications that mirror the original system's kernel-space patterns.

At the user level, Ditto analyzes factors which significantly impact application on-CPU performance. These factors include instruction mix, memory access patterns (data and instruction), branch behavior, and data dependencies [12], [11]. To collect platform-independent metrics, Ditto employs a suite of tools including SystemTap, Intel 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.

D. 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 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. Since relationships between knobs and performance are locally linear, we use a feedback-based heuristic to tune knobs.

E. Implementation

Ditto is implemented primarily in Python and C in about 16,000 lines of code. It supports C/C++ applications, the Apache Thrift and gRPC 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 distributed in a server cluster, using Docker Swarm or Kubernetes. While 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. 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.

IV. Evaluation

A. Methodology

We evaluate Ditto across a diverse set of services, including key-value stores (Memcached and Redis), webservers (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 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 servers run x86 ISA, but differ in the CPU and memory architectures, and their storage and network.

© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3419067

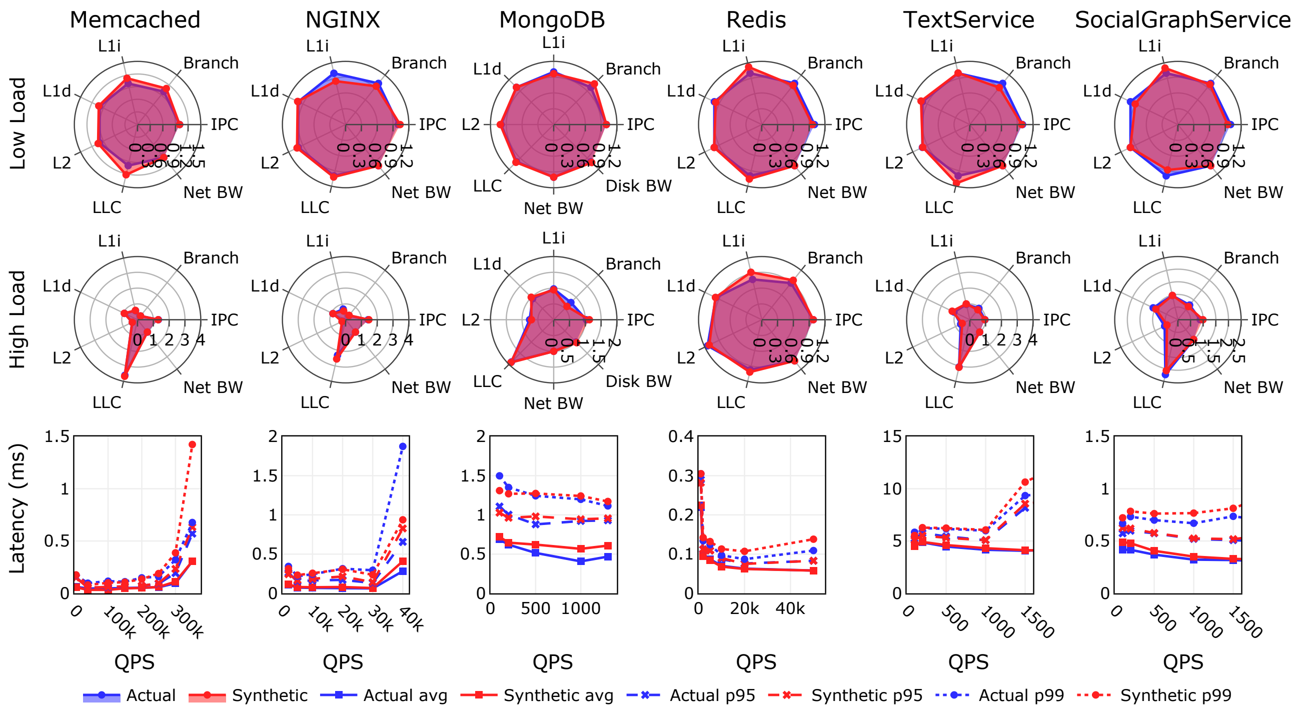

Fig. 3: CPU performance metrics (IPC, branch mispredictions, L1i, L1d, L2 and LLC miss rates), network bandwidth, disk bandwidth (MongoDB only) and service latency under varying load across six services. CPU metrics are normalized to each original application's metrics under low load. Network and disk 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.

TABLE I: Server platform specifications.

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

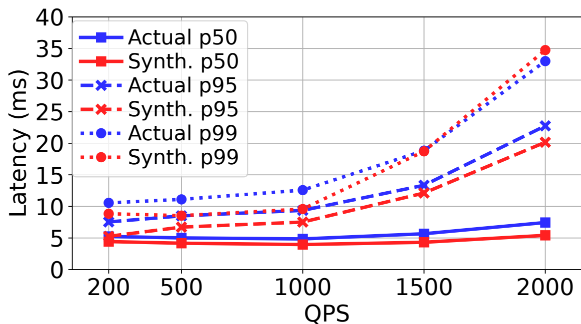

Fig. 4: End-to-end latency for the Social Network.

B. Validation

1) Validation on Varying Loads: Figure 3 shows CPU, network and disk performance metrics, and latency for six applications under different QPS on platform A. In addition to the four single-tier applications, we also show resource characteristics for TextService and SocialGraphService, two applications in Social Network, which are representative of the other tiers of the service. All 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 CPU for the other applications).

The upper two rows show IPC, branch misprediction, L1i, L1d, L2, LLC miss rates, and network and disk I/O bandwidth under low 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, Redis maintains metrics similar to those observed under low load. In contrast, the other five 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 bandwidth also conform to the original by faithfully reproducing the system calls. We

© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3419067

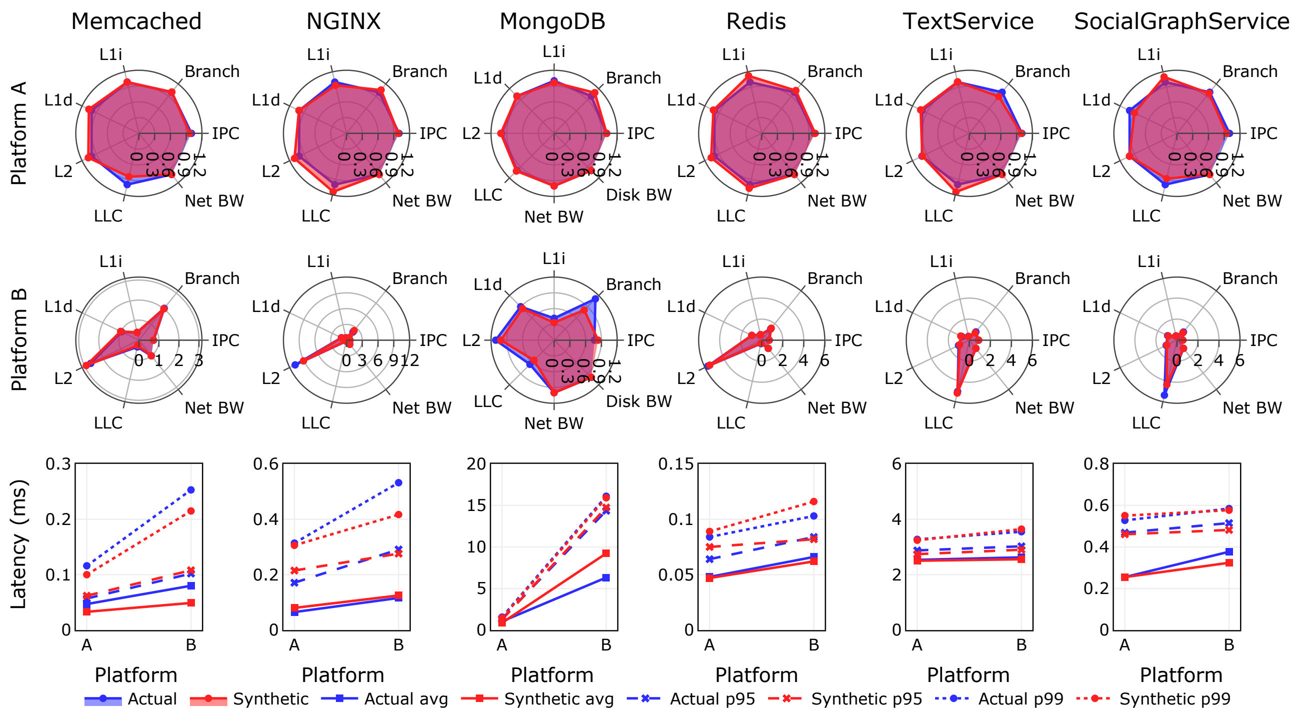

Fig. 5: CPU metrics (IPC, branch misprediction, L1i, L1d, L2 and LLC misses), network BW, disk BW (MongoDB only) and latencies across platforms. CPU metrics are normalized to each original service on Platform A.

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. 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.

Fig. 4 shows the end-to-end latency of original and 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.

2) Validation on Varying Platforms: We validate CPU, network, and disk performance along with service latencies across different x86 platforms. Each application is initially profiled on Platform A, with subsequent validation conducted on both Platforms A and B. Figure 5 shows that the synthetic benchmarks react to platform changes in a similar way to 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. Applications running on Platform B, which is an older CPU generation, have consistently lower IPC. 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 two 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.

C. Case Study: CPU Core and Frequency Scaling

Fig. 6 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.

V. Discussion

End-to-end Application Cloning. Ditto is the first framework to facilitate end-to-end cloning of distributed cloud applications. Though cloning has proven effective for microarchitectural analysis, its broader impact on cloud systems has been

© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3419067

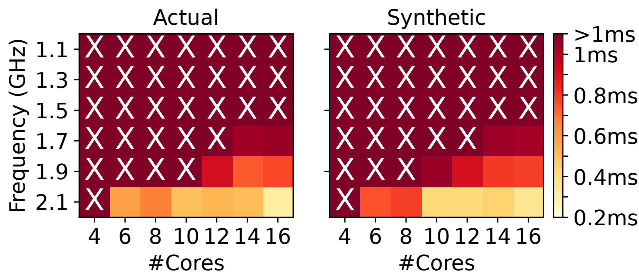

Fig. 6: 99th percentile latency of actual and synthetic Memcached under varying CPU frequency core count.

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 able to easily be replaced 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 "What-if" Simulation: Estimating how a change to an application's design or deployment will impact 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, multi-tier application topologies.

Frameworks like Ditto offer a powerful solution. Consider a microservice provider evaluating a shift from RPC-based communication to message passing. With Ditto, they can seamlessly swap the 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 amount of data transfers over the network would have on performance and resource usage. This simply requires 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. While 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 intellectual property (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.

Acknowledgements

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, an Intel Research Award, an Intel Faculty Rising Star Award, a Sloan Research Fellowship, a Microsoft Research Fellowship, and a Facebook Research Faculty Award.

Mingyu Liang is a Ph.D. candidate in the School of Electrical and Computer Engineering at Cornell University. He works on computer architecture, cloud computing, and new cloud programming models. He is a student member of IEEE and ACM. Contact him at ml2585@cornell.edu.

Yu Gan is research engineer at Google. He received a Ph.D. degree in electrical and computer engineering from Cornell University. He works on the intersection of distributed systems and machine learning. He is a member of ACM. Contact him at yg397@cornell.edu.

Yueying Li is a Ph.D. candidate in the School of Computer Science at Cornell University. She works on machine learning systems, datacenter computing, and computer architecture. She is a student member of IEEE and ACM. Contact her at yl3469@cornell.edu.

Carlos Torres is a Staff Performance Engineer at Meta's Co-Design team. He works on workload characterization and HW/SW optimizations for Meta's large-scale distributed services. Contact him at cltorres@meta.com.

Abhishek Dhanotia is a technical leader whose expertise lies at the intersection of computer architecture and large scale datacenter systems. In his current role at Meta, he leads the work on system architecture, performance and efficiency of Meta's next generation compute platforms. Prior to Meta, Abhishek worked at Intel on server architecture and performance

© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3419067

for multiple generations of Intel Xeon-Phi server processors. Contact him at abhishekd@meta.com.

Mahesh Ketkar is a Principal Engineer and manager in Intel Labs with focus on development of performance analysis and optimization technologies spanning microarchitecture to system-level. He has received two best paper nominations in leading academic conferences and a Mahboob Khan Industrial Liaison Award. He received his Ph.D. 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, where she works on computer architecture and distributed systems. Delimitrou received a Ph.D. degree in electrical engineering from Stanford University. She is a member of IEEE and ACM. Contact her at delimitrou@csail.mit.edu.

REFERENCES

[1] L. Barroso and U. Hoelzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. MC Publishers, 2009.

[2] C. Delimitrou and C. 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.

[3] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 171–172.

[4] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. 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.

[5] B. Gregg, *Systems performance: enterprise and the cloud*. 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 *2006 IEEE International Symposium on Workload Characterization*, 2006, pp. 105–115.

[7] N. Lazarev, N. Adit, S. Xiang, Z. Zhang, and C. Delimitrou, "Dagger: Towards Efficient RPCs in Cloud Microservices with Near-Memory Reconfigurable NICs," in *Proceedings of the Twenty Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2021.

[8] M. Liang, W. Fu, L. Feng, Z. Lin, P. Panakanti, S. Zheng, S. Sridharan, and C. Delimitrou, "Mystique: Enabling accurate and scalable generation of production ai benchmarks," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[9] M. Liang, Y. Gan, Y. Li, C. Torres, A. Dhanotia, M. Ketkar, and C. Delimitrou, "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*, 2023, pp. 222–236.

[10] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*. Portland, OR, 2015.

[11] R. Panda and L. K. John, "Proxy benchmarks for emerging big-data workloads," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 105–116.

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

© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.