Rethinking the Choice between Static and Dynamic Provisioning for Centralized Bare-Metal Deployments

A Dissertation Presented
by

Apoorve Mohan

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Northeastern University
Boston, Massachusetts

July, 2021

Contents

List of Figures

List of Tables

Abstract of the Dissertation

Rethinking the Choice between Static and Dynamic Provisioning for Centralized Bare-Metal Deployments

by

Apoorve Mohan

Doctor of Philosophy in Electrical and Computer Engineering

Northeastern University, July, 2021

Dr. Gene Cooperman, Advisor

Technological inertia leads many organizations to continue to employ static provisioning strategies for bare-metal deployments. This results in architectural and runtime inefficiencies. Yet, most performance and security-sensitive organizations currently employ static provisioning strategies to run and manage their workloads on physical servers (also known as bare-metal servers).

This thesis demonstrates how recent technological advances have given the advantage to dynamic provisioning strategies over static, to improve aggregate bare-metal resource efficiency in centralized bare-metal deployments.

Conceptually, this dissertation has four parts. First, we introduce a new system architecture for dynamic bare-metal provisioning. The proposed architecture reduces the overhead and operational complexity and improves the fault tolerance and performance of at-scale dynamic bare-metal provisioning. To achieve these improvements, it decouples the provisioned state from the bare-metal servers. Furthermore, we demonstrate how the system can also improve bare-metal provisioning performance in security-sensitive deployments.

Second, we identify the performance and operational issues due to intrusive software introspection strategies employed in existing bare-metal deployments. To mitigate these issues, we present a dynamic bare-metal provisioning-based system to enable non-intrusive software introspection of bare-metal deployments.

Third, we describe a novel system architecture to enable cross-framework bare-metal multiplexing in modern data centers. The proposed architecture leverages recent technological advances that enable rapid dynamic bare-metal provisioning.

Finally, we demonstrate how dynamic workload provisioning can improve aggregate throughput of batch workloads hosted on bare-metal deployments.

Chapter 1

Introduction

Many organizations have realized the cost benefits of consolidating their computing infrastructure over managing multiple, relatively smaller compute facilities.

Over the past decade, virtual infrastructure solutions have become the obvious choice in consolidated (i.e., single-tenant managed or centralized) deployments for many organizations due to the scalability, availability, and cost benefits it offers.

However, most performance and security-sensitive organizations such as medical companies and hospitals, financial institutions, federal agencies run their workloads directly on physical (i.e., bare-metal) infrastructure. They are unwilling to tolerate the performance unpredictability and security risks due to co-location and performance overhead or vulnerabilities due to a large code base of the complex virtualization stack.

Besides, more generally, organizations also prefer bare-metal infrastructure when they want to set up their virtualization software stack for offering cloud services (either public or on-premise) and fend off against issues due to nested virtualization, or when their workloads require direct and exclusive access to hardware components (e.g., InfiniBand, RAID, FPGAs, GPUs) that are difficult to virtualize.

Such organizations invest enormous sums of money to buy or rent bare-metal servers and set up and manage bare-metal clusters. Thus, it is imperative that these organizations efficiently operate and utilize the bare-metal clusters they set up to maximize their investment returns.

However, despite the technological advances over the past decade, organizations continually employ *static* provisioning strategies (see Section 1.1) in bare-metal deployments that contributes to poor aggregate bare-metal resource efficiency.

> ***Thesis Statement:*** *This thesis demonstrates how* dynamic *provisioning can mitigate observed inefficiencies in centralized bare-metal deployments. The proposed dynamic provisioning strategies leverages existing storage disaggregation and fault-tolerance technologies to improve aggregate bare-metal resource efficiency. By applying it to four real-world scenarios, this thesis demonstrates the effectiveness of dynamic provisioning.*

1.1 Architectural and Runtime Issues

Over the past decade, there have been continual improvements in data center hardware and software technologies (such as high-bandwidth, low-latency network infrastructure, disaggregated storage solutions, and fault-tolerant distributed and parallel applications). However, past biases towards static provisioning strategies result in inefficient usage of bare-metal resources in centralized deployments. Rethinking the choice between static and dynamic provisioning can significantly improve their efficiency. This section presents an overview of the issues addressed by this thesis.

1.1.1 Lack of Bare-Metal Multiplexing

A long-standing practice by organizations managing their bare-metal infrastructure is to deploy multiple statically sized bare-metal clusters as silos in their data center. Organizations deploy different frameworks (e.g., OpenStack, Slurm, and Spark) in their data centers directly on bare-metal servers due to the guaranteed performance and security advantages provided by bare-metal execution. The workloads running on these bare-metal clusters are known to exhibit heterogeneous resource demand patterns (e.g., diurnal resource usage patterns, long job queues).

Despite complementary resource usage patterns among co-located bare-metal clusters (where the servers of the clusters are attached to a network farbic such that latency and bandwidth are not a issue), organizations do not consider multiplexing bare-metal servers across these deployments to extract the maximum value from the infrastructure. Their reluctance to multiplex bare-metal servers across co-located clusters primarily stems from the complexity and cost of changing the ownership/access of bare-metal servers between clusters and the security and privacy concerns involving bare-metal multiplexing. This leads to low aggregate resource efficiency at the larger organization level.

Multiplexing bare-metal servers across co-located clusters is a complex multi-stage process resulting in multiplexing overheads from tens of minutes to hours. To understand its complexity, let us consider a particular example of a security-sensitive organization multiplexing bare-metal servers between non-trusting bare-metal clusters. In such a case, the multiplexing stages for a bare-metal server would be: (a) disabling the bare-metal server from the orchestration framework managing the cluster (b) backing up any relevant state on the bare-metal server; (c) de-provisioning of the software stack from the bare-metal server; (d) disconnecting the intended bare-metal server from the source clusters networking; (e) scrubbing the bare-metal server for any malicious software firmware; (f) connecting the bare-metal server to the destination clusters networking; (g) booting the bare-metal server and provisioning the intended software stack; (h) re-booting the system with the provisioned software stack while verifying the integrity of the provisioned software; and (i) integrating the bare-metal server into the orchestration framework managing the destination cluster.

To achieve more than marginal gains from bare-metal multiplexing, automation of the multi-stage multiplexing process and the minimization of each stage's overhead is crucial. These stages can contribute anywhere from several minutes to hours towards the total overhead of bare-metal multiplexing when done manually. Such stages include: provisioning and enabling the software stack for the requisite orchestration framework on the bare-metal server; integrating (or isolating) bare-metal servers into (or from) the clusters network domain; scrubbing the local storage on the bare-metal server before being integrated into a bare-metal clusters network domain;

1.1.2 The Need to Substitute Diskless for Diskful Provisioning

The diskful bare-metal provisioning approach employed in data centers has the following issues: high provisioning and multiplexing overheads, and intrusive software introspection.

High Provisioning and Multiplexing Overhead: One of the key contributors towards high multiplexing overhead is how bare-metal servers are provisioned. Many existing bare-metal cluster provisioning solutions install the operating system and application into the server's local storage. Such an installation process incurs long startup delays (tens of minutes to hours) and high networking costs in order to copy large disk images. Moreover, because the user state is local to the server, these solutions lack rich functionality of virtual

solutions, including checkpointing/cloning of images, releasing and re-acquiring servers to match demand, and fast recovery from server failures.

Furthermore, when operating in a security-sensitive environment, the bare-metal servers' state must be verified for any firmware security vulnerabilities. Also, any critical state on the local storage must be copied out and scrubbed off when multiplexing between non-trusting bare-metal clusters. With a diskful provisioning approach and additional security checks, the overhead to multiplex bare-metal servers across clusters increases significantly (anywhere from tens of minutes to hours). Such high multiplexing overheads hampers the bare-metal cluster elasticity and prevents leveraging any short-term resource demand fluctuations across clusters.

Intrusive Software Introspection: Next, introspecting the software stack running on locally provisioned bare-metal clusters for security vulnerabilities in large-scale deployments has several issues. Traditionally, to check for vulnerabilities in the deployed software stack and any malicious entity that co-exist with the running software stack, security solutions have been installed as agents on the systems. These agents run like any other application on the system while implementing their respective security functions. These agents; periodically scan the file system, memory, and running processes; compare their findings against databases of known vulnerabilities or malware; and send reports to the system administrator to summarize their findings. Over the decades, implementing security functions through local agents has become a standard practice in data centers. This model of agent-based software introspection has three critical shortcomings:

- Security agents themselves may become vulnerable and lead to new attack vectors into one's system, as reported through a recent "DoubleAgent" attack that turns one's antivirus into malware or/and hijacks the system. Therefore, there is an increasing need to ensure the agents' own integrity and to sandbox their executions.

- An agent needs to be installed separately on every new system. Thus their operational and maintenance cost increases linearly with the system's scale and becomes challenging at a massive scale. Although automation engines can ease these maintenance overheads, these automation engines, in turn, require running their respective agents on the system.

- While performing periodic security inspections, these agents consume system resources (e.g., CPU cycles, memory, network resources), which may impact the performance

of the primary workloads that the systems are catering to. While the system may tolerate or avoid these overheads, this may, in turn, lead to performance instabilities or resource inefficiencies. This can be serious problem in envrionements such as HPC deployments running performance-sensitive applicaitons and Cloud deployments where 99.9 % tail latency is important.

1.1.3 Poor Throughput due to Static Job Co-scheduling

Besides the aforementioned architectural issues, there is a crucial runtime issue concerning how many-core batch-style workloads are processed on high-throughput bare-metal clusters. Application writers are notoriously poor at predicting how many CPU cores will be needed, and they often declare a requirement for more cores than they need – resulting in wasted CPU cycles. This especially happens when a job execution may pass through several phases, with different resource requirements in each phase.

Several attempts to mitigate this issue employ offline apriori profiling of applications, followed by smart co-scheduling of complementing applications. However, the application set is continuously diversifying, and in the real world, it is not always possible to profile all the jobs.

Furthermore, suppose an organization is hosting a shared cloud service for processing such jobs due to privacy and security reasons. In that case, they will never know about the job characteristics unless the users agree to provide them. In such situations, co-locating such many-cores batch jobs can lead to inefficient usage of the bare-metal cluster.

1.2 Contributions

This section first outlines the research questions answered in this thesis, followed by listing the scientific contributions of this thesis.

1.2.1 Research Questions

This thesis addresses the following questions intended to overcome the issues presented in Section 1.1:

- What are the key considerations (i.e. constraints, benefits, and feasibility) for an organization to enable bare-metal multiplexing across co-located bare-metal clusters running heterogeneous workloads?
- How fast can bare-metal multiplexing in existing deployments be, and how to enable it?
- Can we decouple the provisioned software state from the bare-metal server? If the former can be done, can we do better than agent-based intrusive software introspection?
- Why static co-scheduling single-node many-core jobs batch jobs can be bad, and is there a systematic approach to recover from such co-scheduling decisions?

1.2.2 Architectural and Runtime Enhancements

This thesis presents the design and prototype implementation of systems that follow *dynamic provisioning* strategies to address the research questions mentioned above. The proposed systems leverage existing storage disaggregation and fault-tolerance technologies to enable dynamic provisioning. The proposed systems cater to the following real-world use cases: (a) rapid and secure bare-metal provisioning; (b) non-intrusive software introspection in bare-metal clusters; (c) short-term bare-metal multiplexing across co-located clusters; and (d) adaptive, piecewise co-scheduling in batch clusters.

This thesis makes the following contributions:

1. An architecture and a prototype implementation of a diskless bare-metal provisioning system that leverages existing disaggregated storage solutions to rapidly (de/re-)provision bare-metal servers as fast as seen in the case of virtual machines. The proposed architecture also enables rapid recovery of servers in the case of failures. The improved speed and fault tolerance is achieved by decoupling the provisioned software state from the server to a disaggregated storage.
2. An architecture and a prototype implementation of a system that leverages diskless bare-metal provisioning to enable non-intrusive software introspection in bare-metal deployments. The proposed system architecture reduces the cost of infrastructure required for at-scale software introspection.

3. An architecture and a prototype implementation of a global-optimization-aware cross-cluster bare-metal multiplexing platform. This bare-metal multiplexing platform also enables organization-level optimizations without requiring changes to existing frameworks or workloads, or schedulers. The platform also supports workload elasticity while eliminating the performance and security concerns of virtualization and containerization. This work also includes an analysis of the potential benefits and constraints of bare-metal multiplexing using four real-world production traces from cloud, data analytics, high performance computing, and high throughput computing clusters.

4. A network congestion-control-inspired approach to enable adaptive and piecewise co-scheduling of many-core single-node jobs on a batch processing bare-metal cluster. The performance of the co-scheduled job is monitored periodically using the CPU performance counters and the co-scheduled set of jobs is adjusted using an application-transparent checkpoint/restart mechanism.

1.3 Thesis Organization

The rest of this thesis is organized as follows.

Chapter 2 reviews some background information, motivation, and related work for bare-metal provisioning, intrusive software introspection, bare-metal multiplexing, and batch job co-scheduling. It also presents a discussion on the system feasibility for bare-metal multiplexing in today's data centers.

Chapter 3 provides the details of Malleable Metal as a Service (M2). M2 is a multi-tenant system architecture for provisioning bare-metal servers in a diskless manner.

Chapter 4 overviews the design, prototype implementation, and performance evaluation of a general-purpose approach for Non-intrusive Software Introspection (NSI).

Chapter 5 presents an analysis on the potential benefits, architecture, and prototype implementation of the global-optimization-aware cross-cluster bare-metal multiplexing platform called BareShala.

Chapter 6 focuses on the adaptive and piecewise co-scheduling approach for many-core single-node batch jobs using a network congestion-control-inspired approach called Additive Increase Full Decrease (AIFD).

Next, the concusions and the future directions of this thesis are presented in Chapter 7.

Finally, a detailed system architecture for secure bare-metal provisioning is presented in Appendix A.

Chapter 2

Background and Related Work

This section reviews the existing bare-metal provisioning approaches (Section 2.1), different software introspection mechanisms, and systems (Section 2.2), how bare-metal clusters are set up and hosted in consolidated data centers (Section 2.3), and previous work on job co-scheduling in high performance computing (HPC) and throughput-oriented commodity data centers (Section 2.4).

2.1 Bare-Metal Provisioning

Bare-metal provisioning systems can be broadly classified into two categories, *diskful* and *diskless*, based on where the image is hosted after the bare-metal servers are provisioned.

2.1.1 Diskful provisioning systems

Diskful provisioning systems provision an intended software stack (i.e., the operating system and applications) on the local storage present of the bare-metal servers. The standard provisioning tools used in many bare-metal deployments are diskful. A rich set of open-source and commercial provisioning products such as Emulab [1], OpenStack Ironic [2], Crowbar [3], Canonical Metal-as-a-Service (MaaS) [4], Razor [5], and Cobbler [6] are available for automated diskful provisioning of bare-metal systems. Chandrasekar and Gibson [7] provide a comparative analysis of commonly used diskful provisioning systems.

Diskful provisioning systems can be further divided into two types. The first type of solution automates the manual installation process of the operating system and desired

applications on the local disks (e.g., Redhat Foreman [8]). As they follow a step-by-step installation process, such solutions generally take the longest to provision.

The second type of solution copies a pre-installed image, containing the operating system and applications, onto the local disk over the network (e.g., OpenStack Ironic [9]). The size of such pre-installed images can be tens of gigabytes. Transferring them can overwhelm the network and making them persistent on the local storage of a bare-metal server still requires hundreds of seconds assuming standard hard disks are used.

Both solutions have a lower bound on the time before a server is ready to use, due to the need to reboot twice, once via the Preboot Execution Environment (PXE) to enter the installer, and once to boot into the freshly installed system. Moreover, when using diskful provisioning systems, re-provisioning a bare metal system requires formatting the local disks and then installing/copying the new system. If saving the existing disk state is desirable, the contents of the disk have to be copied away, which again requires hundreds of seconds and further increases the re-provisioning cost.

In general, diskful provisioning systems and the automation tools that employ these provisioning systems (e.g., OpenStack Ironic, Canonical Metal-as-a-Service (MaaS), Redhat Foreman) are designed for setting up long-running bare-metal systems. They consider the high startup delays to be tolerable, assuming that the servers they provision will run continuously for long times. To support fast provisioning and reduce the boot time of diskful provisioning systems, Omote et al. [10] propose BMCast, an OS deployment system with a special-purpose de-virtualizable Virtual Machine Manager that supports OS-transparent quick startup of bare-metal instances.

A fundamental problem with all diskful provisioning systems, is that any modifications to the image are stored on the disks attached to the bare-metal server. This means, for example, that a user cannot easily release and re-acquire servers to match their needs since any state on the local disk is lost when the user releases the servers. Some of the rich functionality users take for granted in virtualized clouds is also not available in these environments. For example, a user cannot snapshot the disk of a physical server and then clone it to boot additional servers. Perhaps most importantly, if a physical server fails, the user cannot easily start up another server from the same disk image or use the disk to diagnose the failure; any state stored on the local disk of the server is inaccessible as long as the server is down. Moreover, in diskful provisioning systems, the local disk-hosted boot drives become a single point of failure. If the disk containing the boot drives fails, the

bare-metal server becomes unusable until the disk is fixed or replaced and data recovery can be daunting in such cases.

2.1.2 Diskless Provisioning Systems

On the other hand, *diskless provisioning systems* keep the provisioning image resident on a network-accessible remote logical disk that appears like a local disk to bare-metal systems. This method of provisioning historically has been used with diskless workstations [11, 12, 13] and HPC systems [14, 15, 16] to boot multiple servers from a single image. The Linux BIOS [17] proposes using Linux as a boot loader to network-boot other operating systems, but that approach requires making changes to the Linux kernel used as a BIOS. Furthermore, diskless provisioning is heavily used in virtualized systems [18, 19, 20].

2.2 Bare-Metal Software Introspection

In this section, we first list the requirements for employing a general-purpose introspection system and then present some existing work on introspection to explain/justify our need for a new non-intrusive introspection solution. We then discuss the advances in technology that enable us to perform agentless introspection in the cloud.

2.2.1 Requirement for Non-intrusive Software Introspection

We first review the key desired features of a non-intrusive software introspection, and how the requirements for supporting those features suggest the design of an agentless solution. In this section, we will briefly discuss those requirements:

Separation of Introspector and Introspected Integrity of security introspectors is critical. To establish confidence in their security assessment (about the introspected system), security introspectors should be impervious to compromises. In the case of an agent-based system, the agents (i.e., the security introspectors) execute as just another application on the system, potentially alongside any malicious software and exposes themselves to compromise [21], doing more harm than good. Moreover, the agents could also transitively be affected by the existing vulnerability in the system. For instance, some Trojan malware could override the *ps* utility on one's system to hide certain processes from reporting, and

then any monitor using that utility to detect suspicious processes on one's system will become ineffective [22]. Hence, it is important that introspectors are separate from the introspected system – i.e., the integrity of the entity performing the security introspection should be verifiable independently from the introspected system.

Managing Introspection at Cloud-Scale It is essential to keep the operational and maintenance cost associated with security assessment in clouds to a minimum. In the existing agent-based approach, a separate instance of security software is required to be installed on every server. The complexity of managing them is directly proportional to the scale of introspected instances in the cloud. For example, introspecting 10K bare-metal instances would require managing 10K agents; and the management complexity would worsen even further if several virtual machines were running on each of the bare-metal instances. Common-routine administrative tasks such as monitoring the health of agents, collection of assessment reports, and rolling of agent updates to all servers become challenging in this approach as the scale grows. Furthermore, when operating at a cloud scale, the networking between introspected systems and the reporting center becomes more complex as well. Therefore, there is a need for a solution wherein the instances in the cloud can scale independently, and the cost of implementing their security introspection can be contained and managed efficiently.

Non-intrusive Introspection Periodic introspection should not have an impact on the performance of the workloads running on the introspected systems. Agent-based solutions require the co-location of the agent with the running workloads. When the agent periodically executes, it contends with the running workloads for resources, potentially causing jitter in the system and impacting the performance of the running workloads [23]. This impact can be mitigated by either: (i) reserving exclusive resources for the agent on each server; or (ii) running introspection when the server is under-utilized. However, both of these approaches have side-effects, since either: (i) the reserved exclusive resources for the agent will remain idle when the agent is not running (leading to resource inefficiency); or (ii) there may be large time windows between introspections, and malicious agents can exploit this feature to avoid introspection. It is preferable to employ an introspection mechanism that does not have a performance impact on the workloads running on the introspected systems.

2.2.2 Introspection Mechanisms

In this section, we present an overview of some of the well-known introspection mechanisms. They can be broadly classified into two categories:

Vulnerability Detection Vulnerabilities are weaknesses that can be exploited by a malicious entity to perform unauthorized actions. Heartbleed (OpenSSL-based) [24], Shellshock/Bashdoor (Unix Bash shell-based) [25], and GHOST (Linux glibc-based) [26] are examples of the most infamous software vulnerabilities seen over the last decade. Various tools such as FlawFinder [27], RATS [28], ITS4 [29], and Foster [30] have been developed to detect software vulnerabilities. These tools employ techniques such as pattern matching, lexical analysis, data flow analysis, taint analysis, model checking, fault injection, fuzzing testing, etc., to detect vulnerabilities [31].

Malware Detection Malware is an ill-intentioned software designed to conceal its identity and cause damage to the computer system that it runs on. Types of malware include Viruses [32], Rootkits [33], Keyloggers [34], etc. Tools such as `chkrootkit` (for detecting the presence of Rootkits) [35], Linux Malware Detect (Linux system scanner to detect threats in shared hosted environments) [36], and ClamAV (an antivirus engine for detecting trojans, viruses, etc.) [37] are examples of well-known malware detection tools. Such tools employ techniques like anomaly-based detection, specification-based detection, and signature-based detection to identify the existence of malware in a system [38].

2.2.3 Towards Non-Intrusive Bare-Metal Introspection

Data center resource disaggregation The advances in data center networking capabilities (such as the common use of 10-40 Gbps NIC's on hosts, full bisection networks, improved Ethernet latency, redundant network switches, higher network bandwidth, link aggregation on bare-metal instances to handle failures [39, 40], etc.) have led to an ongoing paradigm shift towards data center resource disaggregation. By fabricating the same system resource-types on standalone blade servers that are interconnected via a network fabric, high-capacity, low-overhead disaggregated services can be offered [41]. Prominent examples includes, growing preference to use a distributed remote storage over local-disk solutions

for storage for reasons of reliability, cost and scalability [42, 43, 44, 45], ability to improve aggregate resource usage [46], etc.

Unified system for instance provisioning and image management So far, such disaggregations were primarily limited to virtualized systems and were not considered for bare-metal instances. However, solutions such as M2 [47], Bolted [48], OpenStack-Ironic [49], etc., exploit the above-mentioned technological advancements and offer rapid bare-metal provisioning in a fashion similar to virtual-machine provisioning. The bare-metal instances are provisioned from remote disks stored on distributed storage. Due to the aforementioned technological advances, they deliver comparable performance to applications that run over locally mounted disks [47, 50]. Furthermore, they offer image management functionalities such as rapid snapshotting and cloning for the bare-metal instance images. The advent of these systems has exemplified the possibility of a *general-purpose* provisioning and image management system for both virtual and bare-metal instances in the cloud.

2.2.4 Related Work

With the increase in the number, complexity, and code-base size of deployed software systems, the number of threats that can lead to security breaches increases as well [51, 52, 53]. Thus, introspection has become a key requirement for organizational IT deployments in cloud settings [54, 55, 56, 57, 58, 59, 60, 61, 62] and large-scale introspection systems have been developed to address these requirements.

Introspection systems can be classified as one of two types: Agent-based and Agentless introspection systems. In agent-based introspection systems, the introspection agent/software runs on each server and the agent periodically executes the desired introspection mechanisms (e.g., ClamAV [37], chkrootkit [35], Linux malware detect [36], etc.) on the server to forward the introspection results to a centralized statistics collection system. Amazon Inspector [59], IBM BigFix [63], Symantec Endpoint Protection [64], and Tanium Threat Response [65] are examples of agent-based introspection systems.

Agent-based introspection has a number of drawbacks such as being amenable to intrusion and creating interference, and these issues have been previously reported in connection with production deployments [23]. For virtual machines and containers, agentless introspection systems such as OpVis [54], Anchore [66], Clair [67], AquaSec [68],

and Twistlock [69] have been proposed to overcome these drawbacks. These solutions exploit the non-intrusive capabilities available in virtualized/containerized systems that enable snapshotting of the target file-systems/images. Specifically, the open-source tool OpVis [54] snapshots virtualized instances at the host-level and introspects the snapshots using filesystem tree introspection techniques such as Columbus [70]. Anchore [66] and Clair [67] use image-scanning capabilities to introspect container images. AquaSec [68], and Twistlock [69] offer proprietary agentless container introspection solutions.

There exist a few studies that focus on agentless introspection of bare-metal servers that are provisioned to Storage Area Network (SAN) targets. Banikazemi et.al. [71] proposes intrusion detection techniques at the SAN target level. Unfortunately, the introspection they do at the SAN-level leads to interference within the SAN controller's I/O path [72]. IDStor [72] avoids this problem through a network-based intrusion detection approach. It intercepts every iSCSI request between the server and the SAN target, re-creates the filesystem state identifying an inverse mapping between blocks and the inodes of files (external to the SAN), and introspects the re-created filesystem state. However, it cannot detect threats caused by software that has not yet been accessed by the server. Moreover, IDStor approach is not sustainable as it requires developing special-purpose software to identify block-inode inverse mapping for every filesystem type.

2.3 Bare-Metal Multiplexing

Large organizations have realized the benefits of setting up consolidated data centers over separate, relatively smaller, computing facilities for each of its sub-organizations. The set of such organizations is not tiny, and it includes federal agencies, large corporations, academic institutions, etc. Utah Data Center (also known as the Intelligence Community Comprehensive National Cybersecurity Initiative Data Center) is an example of a compute facility shared by various intelligence agencies. Similarly, large corporations such as IBM, Microsoft, and Google host sub-organizations/teams working on different products/services.

Typically, such organizations have complex hierarchical structures, where a top-level entity allocates an infrastructure budget to each of its sub-organizations, and the sub-organizations either set-up a dedicated bare-metal cluster, or rents infrastructure from another sub-organization offerings Infrastructure-as-a-Service on its bare-metal cluster.

The size of these bare-metal clusters is determined by the budget allocated to

a sub-organization – which in turn depends on factors such as the value proposition of the workloads they host, historical utilization patterns, quality-of-service requirements of the hosted workloads, etc. The sub-organizations are required to optimize their allocated budgets. For example, the size of a cluster hosting a cloud-like service should be chosen to support peak utilization demands and not to turn down any incoming users – leading to low average utilization of the available bare-metal servers. On the other hand, a bare-metal cluster hosting a high-performance computing platform is usually limited by the allocated budget – thus resulting in long queues of jobs waiting for available resources to free up.

2.3.1 Target Environments

While several workloads can execute seamlessly on virtualized infrastructure, the core frameworks that provide virtualization (e.g., OpenStack, VMware Vcloud, etc.) prefer bare-metal execution to prevent overheads due to nested virtualization. Realizing the need for bare-metal execution, public cloud vendors have started to offer bare-metal as a service solutions (e.g. Amazon Bare Metal, IBM Bare Metal, Rackspace bare metal, ...) and also have started to partner with companies developing proprietary virtualization technologies to offer cloud services running on top of their bare-metal offerings (e.g., VMware Cloud on Amazon Web Service [73]).

In addition, many performance and/or security sensitive organizations (e.g., high performance computing centers, financial or medical institutions, federal agencies, etc.) also opt for bare-metal execution for their workloads running on frameworks such as Spark, Slurm, etc. Examples of such workloads are, latency-critical workloads that want to avoid QoS violations due to noisy neighbors or virtualization overhead, accelerator-intensive deep-learning and crypto-currency mining applications that cannot tolerate performance overheads of using a virtualized accelerator, I/O-intensive database services that want to avoid dynamic assignment of critical resources such as caches, and network-intensive MPI-based workloads that dislike jitters and overhead incurred using virtual machines.

Running workloads using containers is known to provide near bare-metal execution performance, however, performance and security problems due to co-location, vulnerabilities in the massive trusted computing base of the orchestration software, dependency on the host operating system, privacy/security concerns when running in privileged mode, or the lack of control when privileged modes are disallowed, still render containerized solutions

unappealing to some applications. For example, an organization we work with has different research-groups/teams that are in competition with each other – thus want to hide sensitive information about the product they are working on and hence demand the best isolation possible even inside the organization. Similarly, researchers conducting performance sensitive evaluations also demand strict isolation.

2.3.2 Bare-Metal Multeplexing Feasibility

Time-sharing of bare-metal servers has not been considered so far due to the complexities and security concerns involved with bare-metal time-sharing. Challenging issues such as dynamic bare-metal access management, strict isolation between the frameworks, server verification for compromises, control of hardware configurations (e.g., simultaneous multi-threading, virtualization, hardware-based security, power management, etc.), decoupled decision making at organization-level and framework-level, and the changes required in participating frameworks need to be resolved for rendering bare metal time sharing feasible. In addition to these concerns, some of our partner organizations also communicated their concerns regarding the effectiveness of participating in a bare-metal sharing system i.e. if the gains from their participation would be marginal compared to the time and effort they would be required to put in.

However, given the technological advancements in distributed computing over the past years, we believe that it is possible to tackle the perceived complexities and concerns involved with bare-metal time-sharing. For example, with the advent of software defined networking and system tools (such as OpenStack-Neutron [], Ansible-Networking [], HIL [], etc.) can enable dynamic hardware access management and isolation. Next, data centers have been using remote storage solutions as data drives for bare-metal servers, however, the boot drives still resides in a local storage – slowing down the (de)provisioning of bare-metal server [47, 10, 74]. The advancements in commodity data centers‘ networking and storage solutions has led to an ongoing paradigm shift towards full storage dissaggregation[1]. Solutions such as OpenStack-Ironic [], M2 [47], etc. have enabled rapid (as fast as VMs) bare-metal provisioning by leveraging these techniques. Furthermore, when provisioning a

[1]Common use of 10-40 Gbps network interface cards on bare-metal servers [], the deployment of full bisection networks in data centers [], improved Ethernet latency [], and deployment of redundant network switches in datacenters [] and link aggregation on bare-metal servers to handle failures [] are example such advancements in networking technology, and features like Copy-On-Write, De-duplication, etc. in storage systems have catalyzed this paradigm shift.

bare-metal server statelessly, it is also possible to rapidly ensure if it was compromised by a previous occupant [75] – which is critical when time-sharing bare-metal servers between highly security-sensitive entities. Moreover, to eliminate or reduce the impact of machine failures, distributed frameworks employ various replication and checkpointing based fault tolerance mechanisms to self-stabilize. Time-sharing solutions can exploit these mechanisms to move bare-metal servers from one framework to another – where abrupt removal of a server would be treated as a server failure by the first framework.

2.4 Job Co-scheduling

In HPC centers, people have intensively studied how to co-schedule separate jobs on the same CPU cores of a computer [76, 77, 78]. In particular, if one job emphasizes CPU-intensive computations, while a second emphasizes RAM-intensive computations (e.g., writing sequentially to a large in-memory array), and a third emphasizes I/O, there are substantial savings to be obtained by co-scheduling the three jobs to use the same CPU cores simultaneously.

Similarly, throughput-oriented commodity data centers are highly interested in co-locating jobs. This area of work has seen a recent resurgence due to the presence of many-core computers running a mixture of low-latency server applications (e.g., web searches, commercial transactions, etc.), and background "best-effort" jobs [79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90]. In this case, the low-latency applications have top priority. Background best-effort jobs use available spare cycles, but are killed when the low-latency applications require additional CPU cycles.

Checkpointing has frequently been proposed and tested as a means of eliminating the need for a backfill queue in batch job queues [91]. However, such work takes care to use checkpointing only on a limited subset of the batch jobs. Liu et al. propose the use of checkpointing for improving turnaround times for batch queues [92].

Job scheduling for batch queues (resource managers) has been and continues to be intensively studied [77, 78, 93, 94, 91, 95, 96, 97, 98]. In particular, co-scheduling or co-location of jobs is also actively studied [76, 77, 78, 81].

The issue of co-scheduling or co-location is of special importance in modern data centers. For example, there have been a series of papers, with from Google, concerning co-

location of latency-sensitive applications (such as web searches) with background "best-effort" computations [79, 80, 81, 82, 83, 84, 85, 86, 87].

Chapter 3

Rapid and Secure Bare-Metal Provisioning

3.1 Introduction

Although virtualized cloud services can satisfy the requirements of many applications, some applications still require physical (i.e., bare-metal) servers. Examples include performance or security sensitive applications that cannot tolerate the overhead, unpredictability, and large trusted computing base of complex virtualized cloud services [99, 100, 101], or applications that need direct and exclusive access to hardware components that are difficult to virtualize (e.g., InfiniBand [102], RAID [103], FPGAs [104], GPUs [105], etc.).

Cloud vendors have developed application-specific solutions dedicated to some of these use cases (e.g., Amazon HPC cloud [106], Amazon GPU servers [107], Cirrascale deep learning cloud [108], etc.). However, these compartmentalized solutions lead to cloud silos, reducing the flexibility of the cloud to move resources between different users as demand warrants. Moreover, it is impossible to cover all bare-metal use cases with dedicated solutions; consider, for example, researchers that want to develop their own bare-metal operating system [109], or cloud developers that need to test software on environments identical to the eventual production environments, or applications that are adversely effected by Simultaneous Multi-Threading (SMT)[1].

[1]Cloud providers tend to enable SMT to achieve higher throughput, but not all applications are SMT-friendly when sharing infrastructure [88] – users running such applications may want to control SMT per their requirements.

The demand for bare-metal clouds has resulted in an increasing number of offerings such as IBM [110], Rackspace [111], Chameleon [112] and Internap [113]. These bare-metal cloud solutions install the tenant's operating system and application into the server's local disks. This installation process incurs long startup delays (tens of minutes to hours) and high networking costs to copy large disk images. Moreover, because user state is local to the server, these solutions lack rich functionality of virtual solutions including checkpointing/cloning of images, releasing and re-aquiring servers to match demand, and fast recovery from server failures.

A number of industry and research projects have attacked the performance and functionality challenges of provisioning bare-metal servers [2, 4, 1, 3, 5, 10, 114]; automating the bare-metal provisioning process, reducing the management overhead of the cloud provider, and improving the performance of copying the image to the server's disk. For example, Omote et al. [10] proposed a lazy copy approach that copies the OS image in the background after the operating system is booted using a remote disk. While sophisticated techniques like this can reduce some of the user visible provisioning time, all these approaches end up eventually transferring the boot image to the local disk, and hence still incur overhead to copy the image and have the functionality problems discussed above. Also, the solutions are not designed to service multi-tenant environments where tenants are mutually non-trusting users or organizations sharing a common provisioning service.

We present *M2*, a multi-tenant provisioning system for bare-metal clouds that addresses the challenges described above. Similar to virtualized cloud services, M2 serves user images that contain the operating system (OS) and applications from remote-mounted boot drives. M2 relies on a fast and reliable distributed storage system (CEPH [115, 116] in our implementation) for hosting images of provisioned bare-metal instances and a network isolation service (HIL [117] in our implementation) for isolating tenants in the cloud.

Key contributions of this work include:

1. The definition of M2 a general purpose architecture of a bare-metal cloud provisioning system that exploits remote storage[2] and allows tenants to:

   - rapidly release and then acquire servers to handle fluctuation in demand,
   - rapidly recover from failed servers by booting another server with the disk,

[2]Previous provisioning systems exploited remote storage in special purpose environments, like HPC clusters, where all servers boot the same kernel.

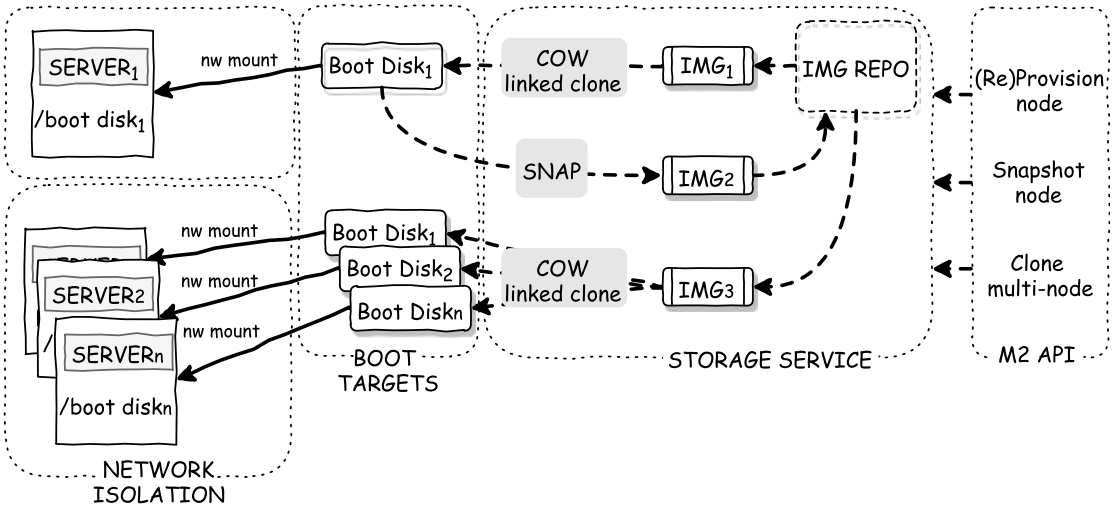


Figure 3.1: M2 conceptual design.

- snapshot and clone disk images,

2. An implementation and analysis that demonstrates that:

   - it is possible to provision and deploy bare-metal systems with overheads similar to deploying virtual machines,
   - performance of the M2-provisioned servers is similar to those provisioned to local disks.[3]

3.2 M2 Design

When designing M2 we first listed the set of features we wanted to have in order to support an on-demand bare-metal cloud service. These features are:

- *Rapid provisioning:* A bare-metal cloud service has to offer on-demand bare-metal servers with minimum startup overhead so that even short-lived deployments with life-spans of a few hours can use bare-metal servers efficiently. If servers take tens of minutes to deploy, the "effective" utilization of the cloud will decrease.

- *Rapid snapshotting, re-purposing, reprovisioning:* Abilities for quickly snapshotting the OS and applications, releasing a server when unused, and being able to quickly provision a server using a previous snapshot are critical for time-multiplexing bare-metal servers

[3] As the focus of this work is to improve the provisioning time M2 only network-mounts boot drives that hosts the OS and applications. Data drives are still hosted on the local disks.

across many users. These features also enable the service to offer "elasticity" to the applications it hosts.

- *Rapid cloning:* The ability to rapidly stand up a large number of servers concurrently using the same saved image is a common request in Infrastructure-as-a-Service clouds. This feature enables easy deployment of parallel/distributed applications and scalability.

- *Support for multi-tenancy:* Existing provisioning tools assume that they are available to just the administrator of the hardware and all of the hardware available in the system is managed by the same entity. However, in a bare-metal cloud service, the provisioning system has to ensure performance isolation and security across its users even during provisioning.

Given the above list of desirable features, we made a set of design decisions for M2. In order to offer rapid provisioning, we opted to use diskless provisioning mechanisms. Using these mechanisms M2 does not need to copy the entire image to the bare-metal server and it can save the overhead to install images and applications to local disks once a cloud image is prepared. M2 can rapidly start running applications by only fetching the necessary OS and application libraries before start-up and further required packages will be fetched on-demand as they are used. Note that the standardization of technologies such as iSCSI [118] has allowed diskless provisioning to be used with commodity servers and clients over any layer-3 network.

We also decided to network-boot the servers from images residing on a distributed storage. M2 can service the images from centralized high performance storage systems using multiple disks to improve boot time. Furthermore, many modern distributed storage systems support capabilities such as *copy-on-write (COW)*, *de-duplication* and *linked-cloning* [119], which are beneficial for capabilities such as rapid cloning and snapshotting.

We note that in diskless provisioning clients are always dependent on uninterrupted access to the centralized storage, and as the number of clients increase, the storage infrastructure has to be adequately scaled to support the increasing load. Network connectivity and availability also plays a critical role in the performance of diskless provisioning systems. However, with the advancements in faster, cheaper and redundant networking (e.g., Clos networks [120]) and storage solutions (e.g., Solid State Drives), datacenter applications increasingly lean towards disaggregating storage services to make full use of the capacity of

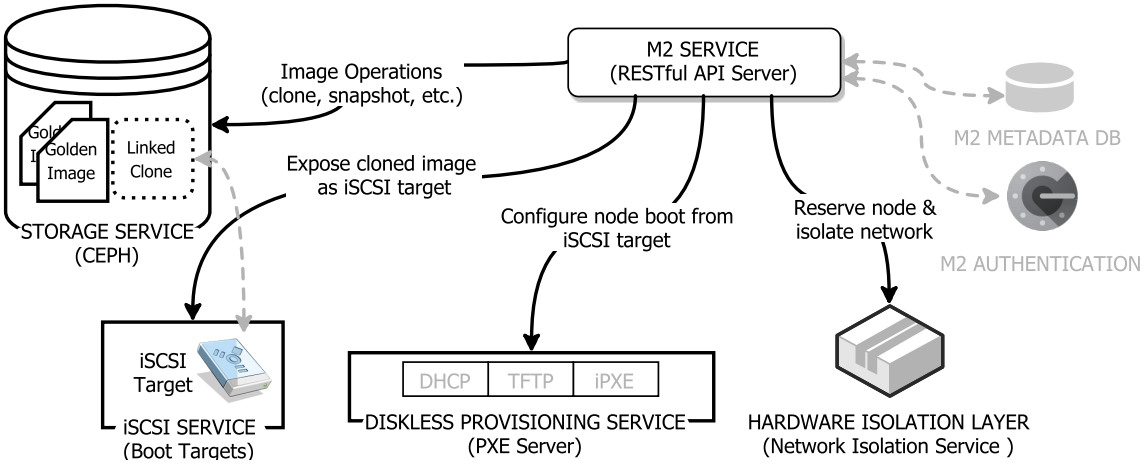


Figure 3.2: M2 components and architecture.

their infrastructures [121]. Diskless provisioning approaches are very much aligned with this trend.

Figure 3.1 presents the conceptual design for M2 and some of the functionalities it offers. As seen in the figure, M2 stores images (user or M2provided) in an image repository. When a provisioning API call is made for a bare-metal server with a given image, a linked-clone of that image is created, followed by network isolation of the requested bare-metal server and mounting of the clone on the bare-metal server. When the server network-boots, it only fetches the parts of the image it uses, which significantly reduces the provisioning time. M2 also supports provisioning multiple servers in parallel from a single image by simply performing parallel provisioning calls.

As seen in Figure 3.1, API call of M2 for creating disk snapshots enables users to create checkpoints/restore-points by saving the current state of the image to the repository and tagging the saved image with a unique identifier. Using linked-cloning and COW, M2 can offer rapid snapshotting.

3.3 M2 Architecture

In this section we discuss our implementation for M2. There are five major components in M2: (i) API Server, (ii) Storage Service (Ceph), (iii) iSCSI Service (TGT Server), (iv) Diskless Provisioning Service (PXE Server), and (v) Network Isolation Service. Figure 3.2 displays these five components. M2 follows a driver-based approach and provides an abstraction for each component. This allows system administrators to replace the solution

used for any of these components. For example, in the current implementation Ceph [122] is used as the storage service, but it is possible to replace it with any other storage service that supports COW like the network-based Lustre [123] or even local systems like ZFS [124], BTRFS [125, 126] or Linux's LVM [127].

Storage Service: Storage Service provides a data store for the cloud images and exposes APIs in order to to rapidly clone and snapshot existing images. In our implementation we use Ceph as our storage solution. Ceph is an open source storage platform that implements a highly reliable and scalable object storage on a distributed cluster [122]. It exposes various interfaces for object, block and file level storage [116]. We used the block storage interface provided by Ceph also known as the Reliable Autonomic Distributed Object Store Block Device (RADOS Block Device, or RBD) to store the cloud images using the *librados* API. *librados* also exposes functionalities such as cloning and snapshotting to manage the RBD-based cloud images. Ceph provides data store and image management capabilities like snapshotting and cloning and offers good read performance [128], which helps in achieving lower latency when M2 tries to fetch the disk blocks on-demand [129].

iSCSI and Diskless Provisioning Services: Our implementation of diskless provisioning is based on network booting bare-metal servers from RBD-based cloud images (stored in CEPH) that are exposed as iSCSI targets. We used the Linux SCSI Target Framework (TGT) [130] to expose the RBD-based cloud images as iSCSI targets. As TGT is a user-space implementation, no extra kernel code is required which improves its compatibility with modified Linux kernels. Being a user-space server also supports multi-tenancy within M2 and defense in depth by enabling the iSCSI server to be run in a Linux container [131], which can permit a single physical server to serve different tenants on different networks without exposing all the iSCSI endpoints to all tenants. TGT also provides native support for RBD-based images, freeing M2 from managing additional mapping state.

Preboot eXecution Environment (PXE) specification provides a standardization for a client-server model for booting servers over the network using Dynamic Host Configuration Protocol (DHCP) and Trivial File Transfer Protocol (TFTP). Although, PXE provides specifications to network boot a server from various targets (HTTP, iSCSI, AOE etc.), it is up to the NIC manufacturer to implement the support to network boot from a particular target into the NIC firmware. As mentioned earlier, M2 uses an iSCSI-based approach to network boot the bare-metal servers. To ensure that M2 can provision any bare-metal server irrespective of the NIC firmware capabilities of that server, M2 first chainloads [132] into

iPXE, which eventually network-boots the bare-metal servers from the exposed iSCSI target. iPXE is an open-source implementation of network booting firmware that provides all the features mentioned in the PXE specifications [132].

The iSCSI Boot Firmware Table (iBFT) [133] gives PXE servers the ability to specify an iSCSI target to which the tenant OS should connect. iBFT makes M2 OS-agnostic, since it eliminates the need for OS-specific parsing and modification of images to configure the identity of the iSCSI server for a given server.

M2 API Server: API Server is a Python-based RESTful web service that controls the flow between different components of M2. Exposed APIs enable users to (de)provision servers, clone provisioned servers, create snapshots of provisioned servers, (de)register users, perform various operations pertaining to images (upload, download, rename, share, list, etc.), list various resources, etc. The API server also maintains a database for various bookkeeping purposes such as maintaining the mapping between bare-metal servers and cloud images, user-cloud image mappings, etc.

While most of the API calls are trivial and trigger various M2 database operations, some of them amounts to interacting with different M2 components — in particular the APIs pertaining to (re/de)-provisioning, snapshoting and cloning servers, and image manipulation. APIs that require interacting with the storage service rely heavily on the performance of the exposed block storage management capabilities.

The provision API enables users to spawn bare-metal instances from existing cloud images hosted in the storage service[4]. It accepts the ID defining the server to be provisioned (e.g. MAC address, NIC Number) and the ID of the cloud image to be used for provisioning as arguments. Upon receiving a provision request, the API server interacts with the Storage Service and creates a linked-clone of the cloud image (passed as the argument to the provision call) and exposes it as an iSCSI target. This is followed by the preparation of the PXE and iPXE configuration files by the Diskless Provisioning Service that will be served to the bare-metal server upon its network boot request. This is similar to how virtual machines are provisioned in different IaaS cloud offerings.

M2 exposes a snapshot API that allows users to create checkpoints/tags by saving a deep copy of the existing server state. Users can use these snapshots to revert back to any previous state in case of a failure[5]. This feature also enables users to manage different

[4]Cloud images need to be registered and uploaded to M2 before the provision API is invoked.

[5]The current implementation of M2 is limited to disk snapshots.

configurations of their servers/clusters. M2 does not expose an explicit API for cloning. Instead users can clone an existing server by creating a snapshot of the current server state and provision one or more new server(s) from that snapshot.

Multi-tenancy and Allocations: For multi-tenancy it is important to segregate each M2 server based on ownership and physically using some network isolation mechanism. M2 uses Hardware Isolation Layer (HIL) [117] for allocations and to achieve multi-tenancy. HIL is a lightweight Python-based layer-2 bare-metal isolation framework that orchestrates allocation of data center compute resources by controlling the networking infrastructure. It exposes an API that enables users to create isolated groups of compute resources from a hardware resource-pool. HIL is a network-switch agnostic framework that follows a driver-based model[6]. HIL is agnostic to the provisioning system running on top of it and is thus our choice for achieving network isolation (multi-tenancy) and allocations.

3.4 Secure Bare-Metal Provisioning

Problem: Bare-metal clouds eliminate the security concerns due to tenant co-location seen in virtualized cloud offerings (as the service unit is an entire physical machine). However, they do introduce the following concerns:

- As existing bare-metal clouds provision physical machines to local storage, the tenant or the provider must scrub any persistent state on the physical machine to ensure data privacy of the tenant releasing the server. However, scrubbing the local storage every time a tenant releases the bare-metal instances significantly increases the overhead to reallocate the bare-metal instances.

- Bare-metal clouds introduce new security concerns (e.g., firmware-level attacks) as the tenants have low-level privileged access to the physical machines. For example, a previous malicious tenant may inject a firmware rootkit to attack a future tenant. Furthermore, if the vendor does not apply firmware security patches promptly, a tenant can become vulnerable to firmware-based attacks.

Solution: All M2 provisioned bare-metal servers are implicitly stateless, with the server volumes accessed on-demand over the network. This approach of diskless provisioning

[6]Currently HIL can manage network isolation for Cisco, Juniper, Dell and Brocade switches.

mitigates confidentiality or denial of service attacks by the provider or subsequent tenants of the server inspecting or deleting a tenant's disk state. As a result of which, if a bare-metal instance is released or preempted, there is no need to scrub the disk (trusting the provider and potentially requiring hours with modern disks). Any persistent state associated with the server can be cleared by simply detaching the network-mounted drive and rebooting the server.

Next, to mitigate the risk of firmware-based attacks, M2's microservices-based architecture can be extended to enable secure provisioning of bare-metal servers. In this thesis's context, secure provisioning refers to comparing a TPM-signed hash measurement of every piece of software (including the firmware, bootloader, kernel, and initramfs) against an allowlist before executing it on the bare-metal server during the provisioning process. A detailed system architecture and performance evaluation for secure bare-metal provisioning by extending M2 was completed as a part of extensive collaboration and is included in Appendix A.

3.5 Experimental Evaluation

In this section we evaluate M2's speed, scalability and performance. We start by presenting our experimental setup. Then we compare the provisioning time of M2 with that of existing provisioning solutions, present the time taken by different M2 API calls, and analyze the scalability of M2. Network overheads associated with using a diskless solutions are also provided and M2's impact on the performance of frameworks and applications is analyzed. We note that in the following experiments, only the boot drives are mounted remotely by M2 since currently M2 focuses on improving boot performance. Whenever data drives are used by applications, those drives are hosted on local disks.

3.5.1 Experimental Setup

In our experiments we used two different environments. In the first environment, each bare-metal server has two 6-core Intel Xeon E5-2630L CPUs (24 cores with hyper-threading enabled), 300 GB 10K SAS HDDs (two servers had 1 TB 7.2K SATA HDDs), 128

[6]Bare-metal servers were provisioned with RHEL 7.1 for all the provisioning systems. The virtual disk size of the image used for both Ironic and M2 was 10 GB. The actual disk size in the case of Ironic was 407 MB whereas for M2 it was 10 GB.

GB RAM and two Intel 82599ES 10 Gbit NICs. A four-server Fujitsu CD10000 Ceph storage cluster with four 10 Gbit external NICs and an internal 40 Gbit InfiniBand interconnect is used as the storage server of M2 in this environment.

In the second environment each bare-metal server has a single Intel 8-core Xeon E5-2650 CPU (2.30GHz, 16 cores with hyperthreading enabled), 64 GB RAM, two 1.8 TB HDDs, and one 10Gbit Ethernet adapter. A ten-server Ceph cluster with a total of 90 spindles and 10GbE internal 40GbE external NICs are used as the storage server of M2 in this second environment.

For both environments, M2's iSCSI and API servers were deployed on a virtual machine with 4 VCPUs and 4 GB RAM. The RHEL 7.1 (or, Centos 6.7) operating system (OS) is installed in cases where an OS installation is performed. By using two different experimentation environments we demonstrate that M2 is hardware-vendor agnostic.

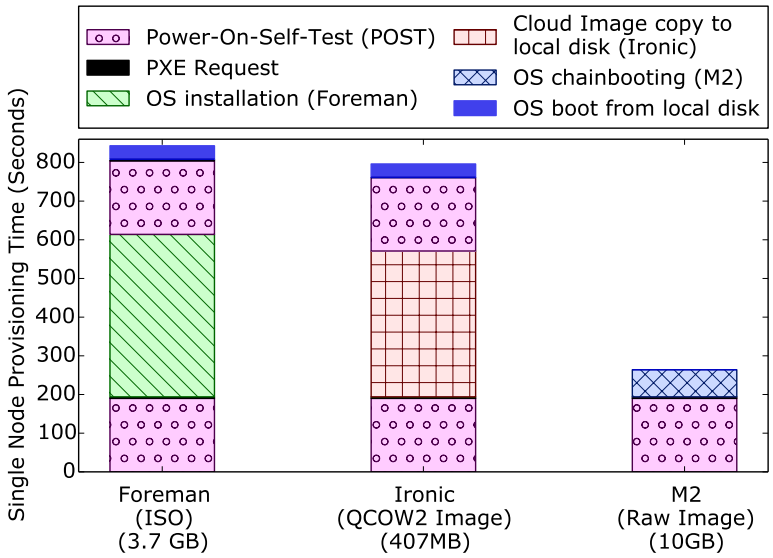


Figure 3.3: Single server provisioning time comparison between M2, Foreman, and OpenStack/Ironic.[7]

3.5.2 Provisioning Time Comparison

Figure 3.3 presents the time comparison of M2 with Foreman, and OpenStack/Ironic, two widely used provisioning systems, when we provision a single bare-metal server in our first environment with a bare RHEL 7.1 operating system. As seen in the figure, the M2

provisioning time is around five minutes. Note that firmware initialization of these bare-metal servers requires more than three minutes; hence half of the M2 provisioning time is spent in firmware initialization. Both Foreman, and OpenStack/Ironic have to go through firmware initialization phase twice. Furthermore, they have to install or network-transfer the OS to a local disk, whereas M2 simply provisions the server out of a remote disk containing the operating system. Due to these advantages, M2 provisions servers around three times faster than both Foreman and OpenStack/Ironic[8].

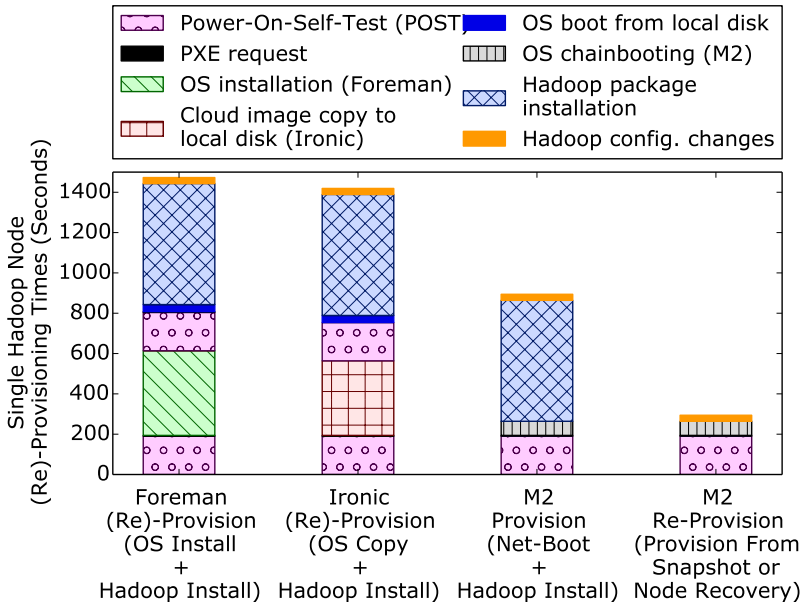


Figure 3.4: Single Hadoop compute server (re)-provisioning time comparison between M2 Ironic and Foreman.

3.5.3 Provisioning Complex Frameworks

Provisioning a server for any framework such as Hadoop or SLURM is a complex and time consuming process handled in many-steps. First, the operating system is installed, and then the relevant packages for the framework are installed, which is followed by making configuration changes to the server. The first three bars of Figure 3.4 show the total provisioning time for a single Hadoop compute server when using Foreman, Ironic and M2. As seen in the figure, M2 can only offer ∼40% improvement during this process as it is dominated by the application installation and configuration.

[8]In Figure 3.3, we do not include the time taken to prepare the provisioning target for the provisioning systems since this is a manual process for Foreman.

Even though installing and configuring frameworks such as Hadoop is a time-consuming process, once a single example setup is made, M2 can leverage its snapshotting and cloning mechanism to safekeep that example and use it for provisioning other framework servers. As shown in the fourth bar in Figure 3.4, with M2, provisioning cloned images that contain desired applications and then doing a final reconfiguration is significantly faster than provisioning servers from scratch.

3.5.4 Using M2 for Failure Recovery

In large datacenter deployments, server failure is a common phenomenon [134, 135, 136, 137]. Recovering from a server failure involves tedious manual operations. If the server is provisioned from the local disk (using Foreman or Ironic), the server becomes unavailable until it is fixed. In addition, if the cause of server failure was disk failure, there is a good chance that all of the user data is lost. In order to re-provision another bare-metal server using Foreman or Ironic as the same Hadoop server, it is required to re-install (or re-copy in the case of Ironic) the operating system and Hadoop packages on the server — leading to a re-provisioning time similar to the provisioning time. As shown in Figure 3.4, the total time to re-provision a single Hadoop compute server is the same as its provisioning time for Foreman and Ironic.

On the other hand, if this server was provisioned using M2, upon failure a new server can be re-provisioned (rebooted) using the image of the failed server that resides in Ceph. The time to re-provision the new server is significantly reduced as there is no requirement to re-install the operating system or any Hadoop packages. As shown in the last bar of Figure 3.4, M2 reduces the re-provisioning time of the servers by up to 5 times as compared to Foreman or Ironic.

Table 3.1: Time required by other M2 operations.

M2 API Call	Time (secs)
De-Provision	6.69
Snapshot	11.65
Clone Image	1.10

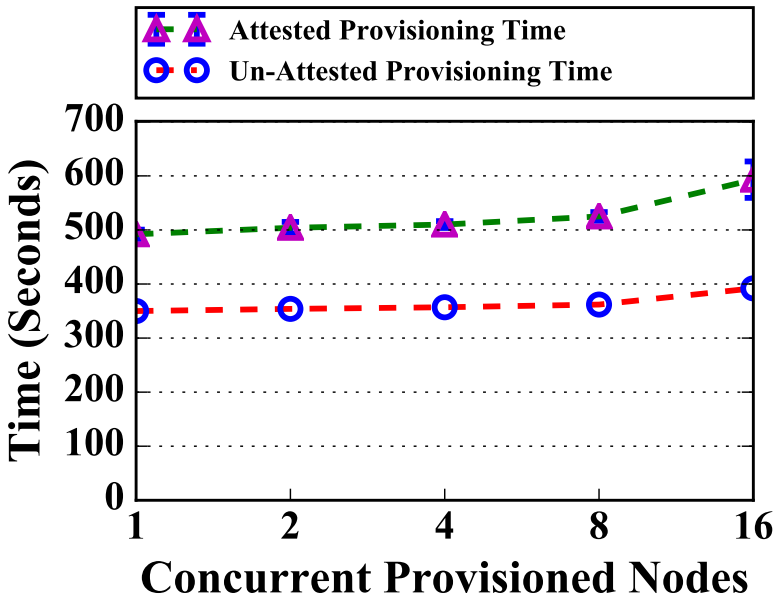


Figure 3.5: M2 scalability analysis.

3.5.5 Operation Times of Other M2 Calls

Table 3.1 presents the time it takes to perform some of the other M2 API calls[9] that require interaction with the storage service. The time taken by the De-Provision operation constitutes the time for those operations pertaining to HIL (detaching the provisioning network from the server), the iSCSI service (disabling the iSCSI target), the storage service (deleting the image associated with the server to be de-provisioned) and the API server orchestration. The time taken by the Snapshot and Clone Image operations are dominated by the storage operations, which include the time taken to flatten a linked clone in the case of the Snapshot operation and the time taken to create a deep copy of a cloud image in the case of the Clone Image operation.

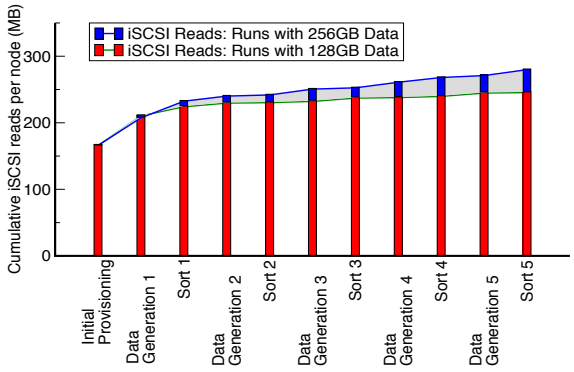
3.5.6 Scalability

In Figure 3.5, we show the time M2 requires for provisioning multiple servers in parallel from our second environment. We increase the number of concurrently provisioned servers from one to 24 and report the time it takes to provision that many servers (upper blue circle line). As seen in the figure, provisioning 24 servers takes only around 20 seconds

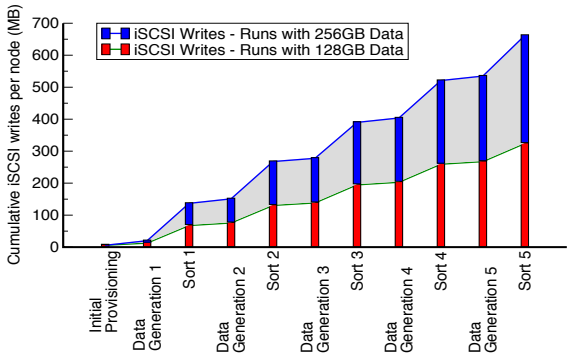
[9]List of all exposed API's can be found at `https://github.com/CCI-MOC/ims/blob/master/docs/rest_api.md`.

longer than provisioning a single server, indicating that even with modest resource usage, M2 is scalable. We observe a slight increase in the M2 overhead (lower line with triangles) since the iSCSI server VM, which has four vCPUs, has to context switch when the number of servers increase above four.

We note here that the current M2 implementation is totally unoptimized and runs multiple M2 services on a single wimpy VM. We expected to see a significant performance degradation in our scalability analysis as requests on M2 increased but observed that not to be the case. As will be shown in the following sections, this is due to the fact that only a tiny fraction of the provisioning image is accessed during booting and application runs and the load on the M2 services is comparatively low.



(a) Read Traffic



(b) Write Traffic

Figure 3.6: Amount of read and write traffic passing through the M2 iSCSI Service hosting the boot drive during provisioning of a Hadoop server and consecutive Hadoop application runs.

3.5.7 M2 Network Traffic Analysis

Figure 3.6 shows the per-server cumulative read and write traffic passing through the M2 iSCSI Service during initial provisioning of a bare-metal Hadoop server and then over five consecutive "data generation and sort" jobs performed over the same server. "Data generation and sort" jobs of 128 GB and 256 GB are performed. Bare-metal servers from the first environment are used during these experiments. The size of the image containing the operating system and the Hadoop packages was 8 GB. Only the boot drive of the server is mounted remotely and the data drives are hosted on the local disk of the server in these experiments.

Figure 3.6a shows that approximately 170 MB of the 8 GB image is read over the network during initial provisioning. Furthermore, both read and write curves flatten after repeated runs, demonstrating that (even with the 256 GB case where the total data handled is substantially larger than the system memory) the file cache is effective at caching the boot drive. After initial boot and application start-up, the sustained read bandwidth incurred is around 3 KB/s; effectively negligible.

Figure 3.6b shows the writes to the network-mounted storage; in contrast to the read case, log writes continue throughout the experiment, at an average rate of approximately 14 KB/s. On further examination, these write target paths such as /var/log, /hadoop/log, and /var/run. (Note that in our deployments, /tmp and /swap are configured to reside on the local disk of servers.) Most of these writes are log file updates made by Hadoop. Although they could be directed to local storage, we did not do so due to their utility for debugging and negligible impact on the data rate.

3.5.8 Performance of M2 Provisioned Systems

In this section we examine the impact of M2 on the performance of applications and frameworks that generally run on bare-metal servers. To this end, we compared the performance of these applications and frameworks when they run on top of M2-provisioned systems (network-mounted) and systems provisioned via Foreman (installed from a local disk).

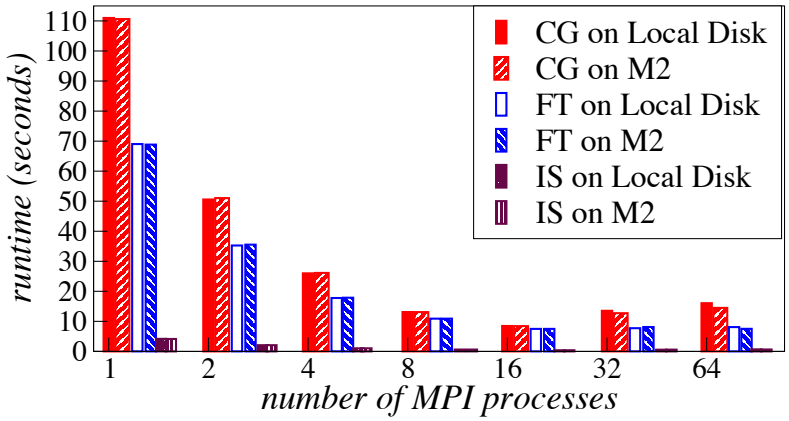


Figure 3.7: M2 and local-disk runtime performance comparison of HPC applications (Conjugate-Gradient (CG), Fourier Transform (FT), and Integer Sort (IS) benchmarks from the NAS suite [138]).

3.5.8.1 HPC Applications Runtime Performance

In Figure 3.7, we compare the runtime of HPC applications (FT, CG, IS) from the NAS Parallel Benchmarks (NPB) [138] running on Foreman-provisioned (installed from local disk) and M2 provisioned (network-mounted) clusters. We ran these benchmarks in our second environment. NPB is a set of programs designed to evaluate the performance of parallel supercomputers. Three benchmarks (i.e., FT, IS and CG) with distinct behaviors were used to evaluate the system. IS performs random memory access, CG has an irregular memory access and communication pattern, and FT does frequent all-to-all communications. We used class B of the MPI version of NPB. Each benchmark was compiled to run with 2^n processes where $n \in \{1, \ldots, 8\}$. Each build was executed using Open MPI on local and remote installations.

As shown in Figure 3.7, almost equal execution times were noted in the case of both M2 and Foreman, resulting in similar height bars. The results indicate that M2 and diskless provisioning have no additional overhead when executing CPU- or memory-intensive HPC jobs. Note that HPC applications already perform well with remote boot drives as such solutions are frequently employed in Beowulf clusters and supercomputers. Hence good performance of M2 is expected in this scenario.

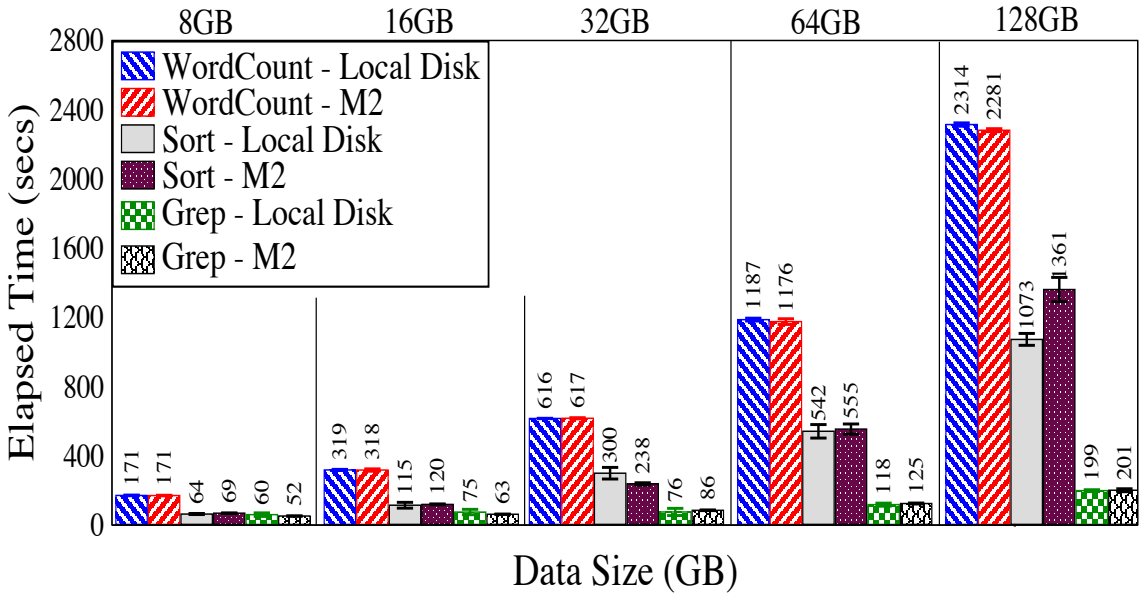


Figure 3.8: M2 and local-disk runtime performance comparison of standard Hadoop benchmarks (WordCount, Sort, Grep).

3.5.8.2 Hadoop Runtime Performance

To measure M2's performance under high network and disk I/O usage we tested its performance when it runs Hadoop jobs. We performed a series of experiments on an 8-server Hadoop cluster as we varied the data set size between 8 GB, 16 GB, 32 GB, 64 GB and 128 GB. We used the first environment for these experiments. Figure 3.8 compares the runtime of standard Hadoop benchmarks (Sort, Grep, WordCount) running on clusters installed from local disks and from network-mounted clusters. In both cases, data disks hosting the Hadoop Distributed File System reside on local disks of the servers. Reported numbers are the average of five runs. We observe that deviations among runs on the same configuration are negligible.

As shown in Figure 3.8, the difference in runtime performances of local-disk-installed and M2-provisioned systems are negligible, with the exception of the Sort experiments for 32 GB data and 128 GB data. We hypothesize that this exception may be caused by the non-deterministic behavior of random sorting benchmarks. The "good" performance of M2 justifies our hypothesis that even for applications that create a significant amount of network traffic and disk I/O, the performance of the application is not adversely impacted by remote

mounting the boot drive.

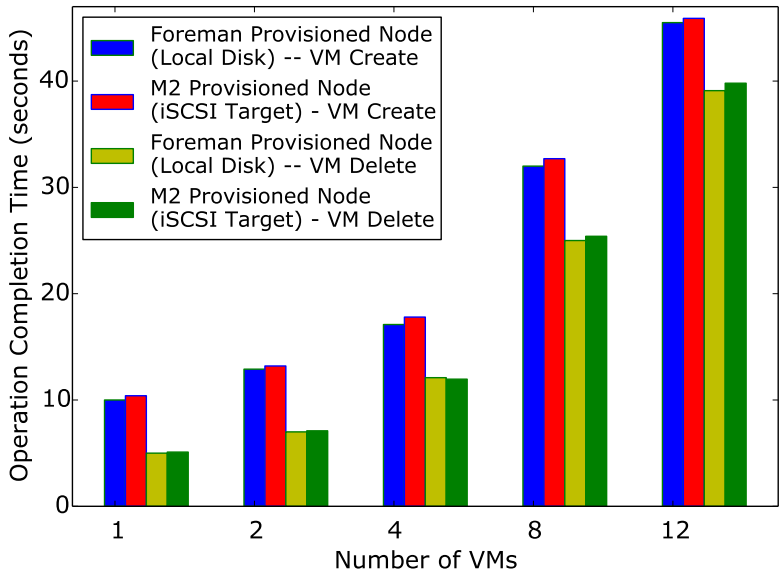


Figure 3.9: OpenStack operation performance comparison between Foremen and M2-provisioned servers.

3.5.8.3 OpenStack Operations Performance

Cloud management systems such as OpenStack that offer virtualized services are also generally deployed on bare-metal servers. In this experiment, we set up OpenStack-based clouds to run on top of M2-provisioned and Foreman-provisioned systems in our second environment. We measured the performance of the two virtual machine operations, namely VM create and VM delete, in these two setups. We used the Rally benchmarking tool [139] for these experiments, varying the number of parallel operation requests issued between 1 and 12. As shown in Figure 3.9, negligible performance degradation was observed for both creation and deletion operations between Foreman- and M2-provisioned servers.

3.5.8.4 Latency and Throughput of Database Operations

Due to their stringent performance requirements database systems are commonly deployed over bare-metal servers. To test if M2-provisioned servers can provide satisfactory performance while running database applications, we compared the latency and throughput of various database operations when running commonly used databases on servers provisioned

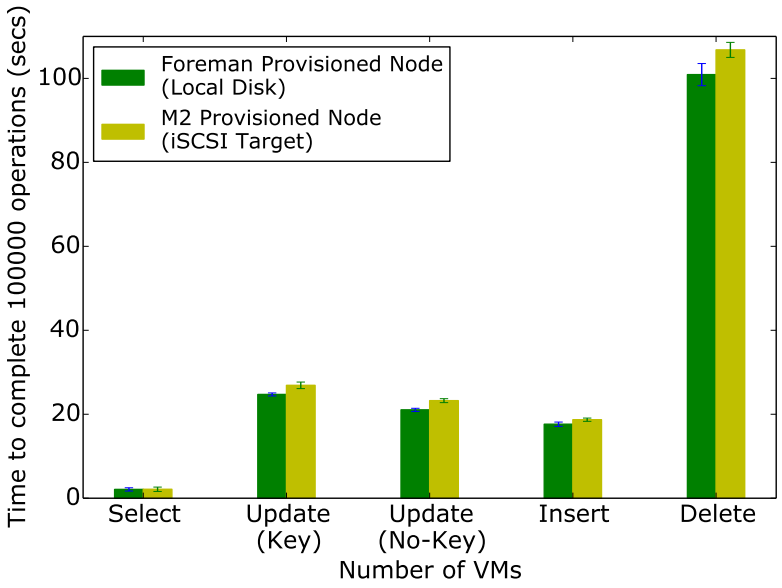


Figure 3.10: MariaDB operation latency comparison between Foremen and M2-provisioned servers.

via M2 versus Foreman. Note that, the data disks holding the actual database data is hosted on local disks in both cases.

Figure 3.10 compares the latency of database operations when running the popular MariaDB database on a server provisioned via M2 versus Foreman. This experiment was performed using the Sysbench benchmarking tool [140] in our second environment. Multi-threaded Online Transaction Processing (OLTP) tests for Select, Update, Insert and Delete operations were performed on the default "sbtest" [141] table generated by Sysbench with one million rows with InnoDB as the storage engine for MariaDB. The number of select operations executed during the test was 100,000, whereas 10,000 operations were executed for each of the update, insert and delete operations. The update operation test had two versions — updating an indexed column (Key) and updating a non-indexed column (No-key). For each test, the number of threads was fixed at 4.

As seen in Figure 3.10, there is a negligible impact on the latency of the select operation for MariaDB when running on an M2 provisioned system. In contrast, in the case of update, insert and delete operations, we observe about 4% degradation in the case of M2 provisioned servers.

Figure 3.11 compares the MariaDB read and write throughput when it runs on a

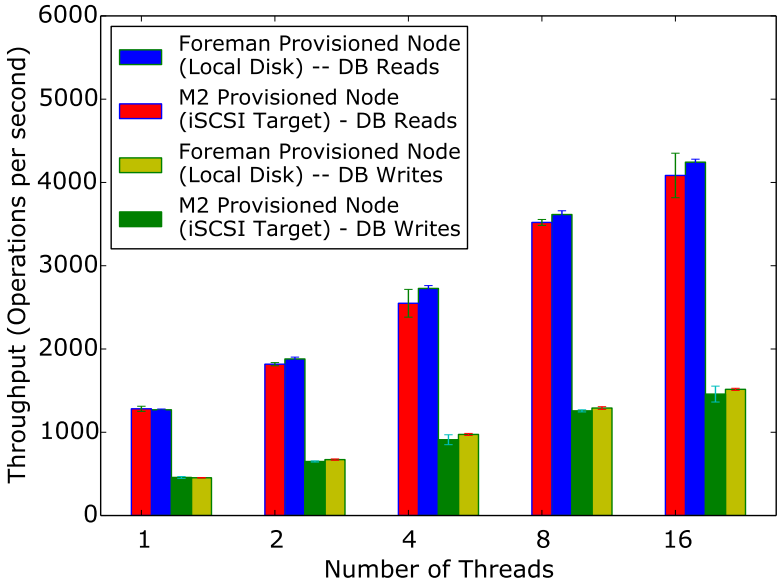


Figure 3.11: MySQL read/write throughput comparison between Foremen and M2-provisioned servers.

server provisioned via M2 and Foreman. This experiment was also performed using Sysbench. In this experiment, we measured the total number of random reads and random writes performed in 300 seconds — varying the number of threads. The throughput of both random reads and random writes in the case of M2-provisioned servers was either at par with their Foreman-provisioned counterparts or saw a degradation of less than 5%. These experiments were also executed in our second environment.

The results in Figure 3.10 and Figure 3.11 indicate that the impact on database performance of remote-mounting the boot drive is less than 5%. This is potentially due to the excessive system memory use by the database, which potentially leads to a competition between the OS and the application for memory pages. Considering the additional benefits M2 offers such as easy fault recovery and easy backup, we believe many deployments will find this additional impact tolerable.

Chapter 4

Non-Intrusive Bare-Metal Introspection

4.1 Introduction

Many organizations deploy security agents alongside the provisioned software stack to verify sanity, i.e., to ensure that the software stack is not compromised. These agents run just like any other application on the system while implementing their respective security functions. IaaS-cloud providers even offer deployable agent-based introspection solutions [59] and image templates already "baked" with such security agents [142][143], which report to a provider-managed central monitoring service. These agents periodically scan the file systems, memory, and running processes of the systems within their scope in order to compare their findings against databases of known vulnerabilities[1] or malware, and send reports to the system administrator summarizing their findings.

Over the decades, these practices of implementing security functions through local agents have become a standard in data centers and clouds, but this approach has several shortcomings (see Fig. 4.1). *First*, security agents themselves may become vulnerable and lead to new attack vectors into one's system, as reported through a recent "DoubleAgent" attack that turns one's antivirus into malware or/and hijacks the system [21]. *Second*, as an agent is required to be installed separately on every system, the operation and maintenance of

[1]Non-profit organizations (e.g., Mitre Corporation [144], National Institute of Standards and Technology [145], etc.) maintain and continuously update open-source databases of vulnerable software and software configurations for public use.

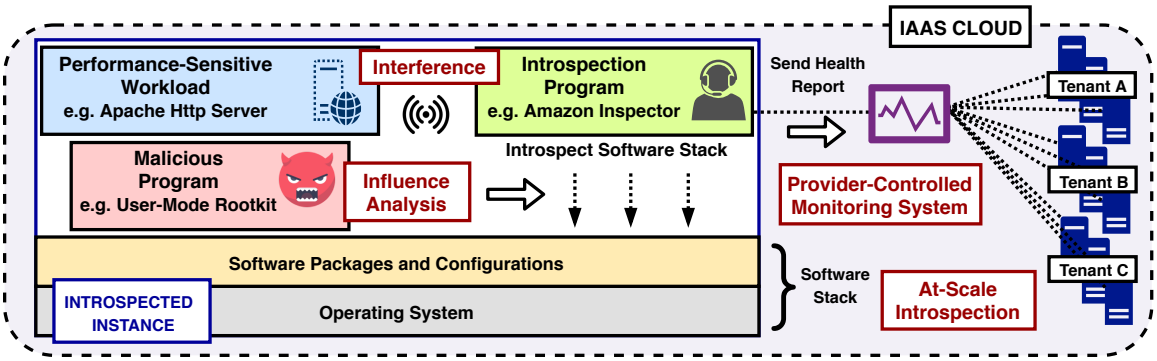


Figure 4.1: Visualizing agent-based introspection.

these agents becomes challenging at cloud-scale. For example, introspecting 10K bare-metal instances would require deploying and managing the same number of agents, and a virtualized environment with 64 virtual machines per host would require 640K such agents. *Third*, while performing periodic security inspection, these agents consume system resources (e.g., CPU cycles, memory, network resources, etc.), which may impact the performance of the primary workloads that the systems are catering to. While the system may tolerate or avoid these overheads, this may, in turn, lead to performance instabilities or resource inefficiencies [146]. *Finally*, a provider-managed introspection service is intrusive for privacy and security-sensitive organizations such as federal agencies, financial institutions, hospitals, etc. Despite the economic benefits of operating in a public cloud, they refrain from using public cloud offerings due to the lack of control and trust in the provider. For such organizations to use public cloud offerings, they would want to enjoy private datacenter-like properties (e.g., control, privacy, security, etc.), even when operating in a public IaaS-cloud [147].

Several efforts have been made to overcome the shortcomings of agent-based introspection [71][72][148][149][54] for specific instance types. But unfortunately, none of these offers a solution that is *non-intrusive, general-purpose*[2] *and tenant-controllable* at the same time. These efforts either intercept I/O requests of an instance provisioned to a remote virtual disk and recreate the filesystem state to perform introspection, or they propose provider-managed solutions to snapshot virtual machines at the host-layer and use the snapshot to perform introspection. For example, Banikazemi et al. [71] propose intrusion

[2]In the context of this work the phrase *"general-purpose"* refers to instance-type (i.e. virtual machine or bare-metal server) and operating-system agnostic. Furthermore, a general-purpose introspection system should not require specialized hardware (e.g., SAN) or support from the hypervisor (in case the of virtual machines).

detection techniques for instances provisioned to a SAN target; however, this can lead to interference within the SAN controller's I/O path – this not only violates the requirements for non-intrusive introspection but also requires specialized hardware. In another proposal, Richter et al. [148] propose recreating the disk state of virtual instances by intercepting the I/O between the hypervisor and the disk emulator, which is not a general-purpose solution as it would not work for bare-metal instances. Oliveira et al. comes closest to achieving non-intrusive introspection by snapshotting running virtual machines and exposing the snapshots as read-only pseudo-devices for out-of-band introspection, however, not only is this approach instance-specific; it still requires resources and access to the host where the VMs are running. Furthermore, these systems require the tenant to trust the cloud provider fully.

In this work, we explore the answers to the following questions:

- *Can we develop an agentless introspection technique that works for both virtual machines and bare-metal servers?*
- *Can we develop a system that has limited complexity for small-scale deployments, so it can be deployed in private clouds and modest clusters?*
- *Can we reduce the overhead and complexity of security introspection for cloud-scale systems?*
- *Can we support tenants that don't want to trust the provider (e.g., those that encrypt their storage)?*

Disaggregated storage allows us to address these questions. The most critical aspect of non-intrusive introspection is to have continuous and non-intrusive access to the state of the provisioned software stack. Disaggregating the persistent state of a running instance to a distributed storage system (e.g., Ceph [150], Lustre [123]) provides a clean separation of the provisioned software stack from the running instance. This disaggregation also enables a simple and practical mechanism for non-intrusive access to the state of the provisioned software stack, which can be exploited by different introspection and inspection APIs to overcome the shortcomings mentioned above for agent-based approaches. Until recently, storage disaggregation has been primarily limited to virtualized instances. However, recent research has demonstrated the possibility to use storage disaggregation for provisioning bare-metal instances with negligible performance overheads [151][47][48][152][39][42][153][154].

In this work, we present the design and prototype implementation of a general-purpose approach for Non-intrusive Software Introspection (NSI). NSI exploits the capability offered by distributed storage to create a cheap copy-on-write snapshot from a remote server and mounts the snapshot as a read-only volume on the remote server to introspect the latest state of the provisioned software stack. This approach has several advantages over previous ones. Since it uses a standard OS with support for arbitrary file systems, we can mount any volume, requiring none of the special purpose techniques developed in the hypervisor or SAN. Moreover, as the OS of the introspecting server is isolated (not exposed to the internet), it is simpler to ensure that the introspecting system is not compromised. Furthermore, the introspection is now decoupled from the tenant instance and is running on a remote server, thus avoiding performance impact on the workloads executing on the tenant's instance. NSI also enables tenants to create their own introspection servers to perform introspection on volumes encrypted by tenant-controlled keys, avoiding the need to trust to the provider; the only capability this requires is to ensure that the tenant has a way of mounting her own volumes.

Key contributions of this work include:

- We propose the design and prototype implementation of NSI, a general-purpose approach for non-intrusive software introspection. We also present a brief discussion on other security analytics use cases that can be aided by NSI. NSI consists of a set of microservices, reducing its deployment complexity and enabling it to scale-out or scale-up quickly. By opting for NSI's modular design, IaaS-providers can enable security-sensitive tenants to control and verify components required to introspect their provisioned software stack.

- A prototype implementation for NSI utilizing open-source components is provided[3]. NSI's modular microservice-based structure lends itself to replacing the underlying components – enabling system administrators to replace any underlying component to keep up with the changing technology. The prototype implementation is straightforward. We implemented introspection capability into an existing bare-metal provisioning system. All we needed was a way to snapshot and mount volumes. Although our prototype implementation is based on a bare-metal provisioning system, it works with virtual machines as it is.

[3] `https://github.com/CCI-MOC/abmi`

- Our prototype implementation shows that the complexity of performing non-intrusive software introspection is small; i.e., all needed functionality can be contained in a simple set of services deployed in a VM. Deployment is straightforward for an enterprise and public clouds, and it can scale up or scale out quickly with small management overhead. Our evaluation shows that one can dedicate a modest amount of infrastructure to introspect on many computers; e.g., in our analysis, one system with 32 cores, 64 GB memory, and 10 Gbps NIC can perform basic software introspection every 5 minutes for up to 463 other computers. Based on the evaluation results, we deduce that for a 10K server cluster (i.e., a typical Borg cluster [155]), it would take 23 servers to perform basic software introspection, which is a trivial amount of infrastructure to set up and manage, rather than reserving resources for and coordinating with introspection agents running on all 10K servers.
- We also evaluate the performance impact of NSI's prototype implementation on real-world applications running on introspected instances and compare it with agent-based approaches. While agent-based introspection can lead to up to ∼12% performance degradation NSI is observed to be scalable with a negligible impact on the performance of the workloads. Our evaluation also demonstrates the possibility of reserving resources with high confidence for a particular introspection mechanism even across different applications.

4.2 NSI Overview

This section presents an overview of NSI's design, components, and workflow; and our prototype implementation. We also briefly discuss some of the other use cases that can be addressed using the proposed design.

4.2.1 Design Philosophy

The key goals of NSI are: (1) to enable non-intrusive software introspection in the cloud; (2) the introspection solution should be agnostic to cloud instance type (i.e., virtual machine or bare-metal); and (3) cloud providers should be able to support security-sensitive

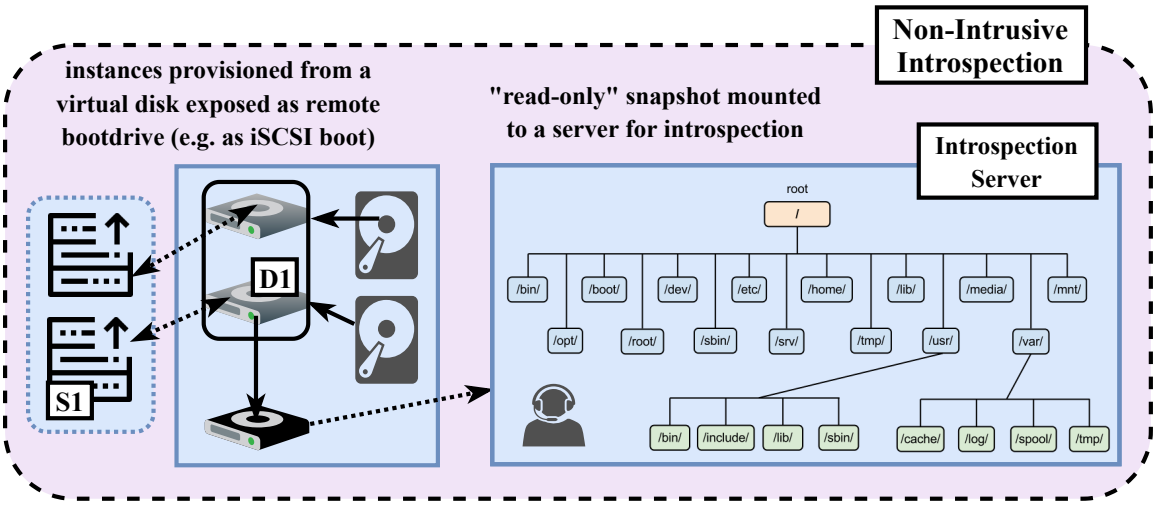


Figure 4.2: Non-intrusive introspection of instance S1 provisioned to virtual disk D1.

tenants. These goals have a number of implications in NSI's design. Fig. 4.2 present a high-level overview of of the proposed non-intrusive introspection.

NSI relies on diskless provisioning of cloud instances to access the state of the provisioned software stack. By doing so, NSI is not required to execute an agent on the introspected bare-metal instance or the host (in the case of virtual machines). Furthermore, using the feature-set offered by today's modern distributed storage systems, NSI can discreetly create lightweight read-only clones of the current software stack, which can be accessed remotely without interfering in the I/O path of the remotely provisioned instance. The read-only clone can be mounted remotely and be introspected non-intrusively – irrespective of the instance-type. Modern distributed storage implementations expose block device interfaces for operations such as lightweight cloning, remote mounting, etc. By provisioning instances in a diskless manner, tenants workloads are decoupled from the software stack, opening the possibility for discreetly snapshotting the current state of the software stack and performing out-of-band introspection on the latest snapshot. Thus, while performing periodic introspection, NSI avoids any performance impact on the workloads executing on an the introspected instance. NSI opts for a microservice-based architecture for software introspection to enable independent control and management of different components required for software introspection. Tenants can either rely on the provider to implement all these components or bring in their own implementation of a particular component. Tenants can

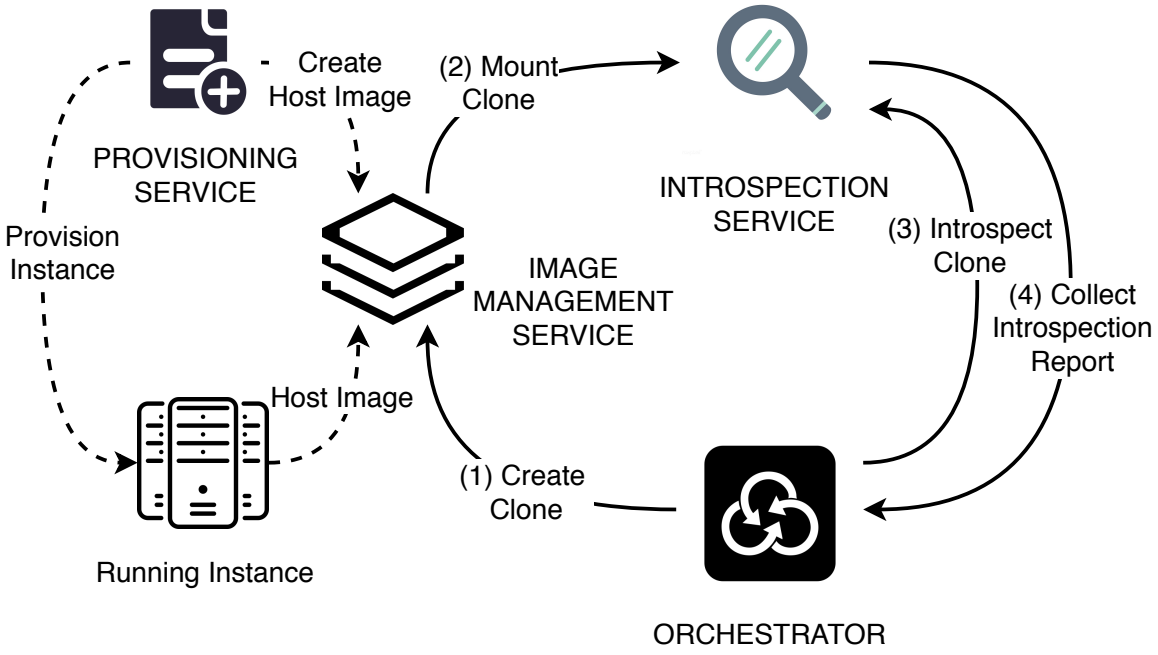


Figure 4.3: NSI workflow design

also set up their own introspection servers to perform introspection on volumes encrypted by tenant-controlled keys, avoiding having to trust the provider; the only capability this requires is to ensure that the tenant has a way of mounting her own volumes.

4.2.2 Components and Workflow

NSI consists of four microservices. Fig. 4.3 presents the workflow design for NSI between the four microservices are: (a) Provisioning Service, (b) Image Management Service, (c) Introspection Service, and (d) Orchestration Service.

Provisioning Service This service provisions cloud instances from the remote-mountable image disks made available by the Image Management Service. For newly provisioned instances, the service calls the Image Management Service to create a new host image, and then the cloud instance is provisioned from that host image. A server can be released by a tenant, and later the tenant can restart the server by pointing it to the same image. We call

this re-provisioning. If the instance is being re-provisioned, the Provisioning Service uses the existing image hosted at the Image Management Service for re-provisioning.

Image Management Service This service hosts the remote-mounted images for cloud instances and provides APIs to rapidly snapshot or clone a provisioned cloud instances‘ image. The remote provisioning service heavily relies on the image management service to perform its activities, but NSI mainly uses the cloning capabilities of this service to clone the host image of the instance to be introspected.

Introspection Service This service first mounts a clone (lightweight snapshot) created by the Image Management Service for the instance to be introspected, mounts the clone as a as a standard filesystem, and then scans the mounted image to check for vulnerabilities. It maintains a database of known vulnerabilities. This database is populated and periodically updated by querying vulnerability databases (open- or close-source based on availability). The introspection service can run rootkit analysis and/or software vulnerability analysis over the mounted volume. For security-sensitive tenants, this service should also have a mechanism to manage secrets (encryption keys) to be used when mounting encrypted disks. We note that introspection operations that need memory analysis are not immediately supported with this design.

Orchestration Service This service controls the remote introspection workflow among the different components. It also offers an interface for tenantss to introspect the instance they own. Tenants can submit requests to this interface for introspecting the instance they control. This allows them to control and change the introspection periodicity on demand, and hence it allows them to control the cost of introspection of their instance.

As shown in Fig. 4.3, when performing a remote introspection, the Orchestration Service: (1) creates a clone of the provisioned instances‘ remote disk via the Image Management Service; then (2) mounts the filesystem present on the clone; and (3) invokes the Introspection Service to perform vulnerability analysis on the mounted image. Finally, (4) it collects the vulnerability analysis results and reports the data back to the tenant that initiated the introspection.

4.2.3 Extended Scope

NSI is an extensible system that can be used to support the enforcement of various Security and Compliance Industry Standards. This includes government regulatory standards like FedRAMP [156], NIST [145], and other industry standards like PCI-DSS [157], Center for Internet Security (CIS) [158]. These are IT/Cloud standards required across Financial, Payment Card industries. While we believe many of these requirements can be implemented on top of NSI, in the scope of the work, we discuss *two* specific requirements that can be satisfied by our system.

Configuration Analytic Software misconfiguration has been a major source of availability, performance, and security problems. In a virtualized and multi-tenant cloud environment, it is non-trivial to ensure correctness when configuring and cross-configuring such components. Moreover, configuration checks are also part of the different regulatory compliance obligations [158] [145] [156]. There are existing agent-based solutions [63] [159] to facilitate configuration management for VMs and bare-metal systems, and there are also agentless solutions [160] [161] [162] for containers. The unique storage dis-aggregation framework in NSI allows us to bring agentless configuration analytic capabilities for VMs and bare metals to scans application and system configuration settings to test them for compliance and best-practices adherence from a security perspective.

Integrity Assurance Another critical security capability for the system is to identify the tampering of critical system management artifacts. These artifacts include configuration files, credentials, secrets-keys, or any other confidential data from your system. This is again regulated under the Chapter 11.5 of PCI-DSS standard. By periodically introspecting the system state and comparing it to the previous state(s), we can identify the file modifications in the system. These file modifications are further semantically analyzed to determine higher-level user action causing the change. For example, if we identify "/etc/passwd" and "/etc/shadow" files are modified, by comparing the content change, we can determine user add/remove action in the system. These changes are then compared against whitelisted actions to realize if unauthorized actions are triggered in the system. State-of-art integrity assurance solutions requires an active monitoring agents like $inotify$ or $auditd$ into system. Operating these agents in context has proven to be detrimental to the application performance.

The agentless framework in NSI can be very efficiently leveraged to provide the integrity compliance assurance for VMs and bare metals.

4.2.4 Prototype Implementation

Since NSI follows a service-based approach, it allows administrators to replace the solutions used for any of the underlying services with the solutions that they prefer.

In our implementation, we employed M2 [47] as the Provisioning Service, and M2 in conjunction with Ceph [163] as the Image Management Service.

M2 is an open-source, multi-tenant, diskless provisioning service. It provisions instances to a remote disk residing on an image store backed up by a distributed file system, and uses the Hardware Isolation Layer [164] tool to isolate the provisioned servers. In our implementation, images of the instances provisioned by M2 reside in Ceph. M2 employs an iSCSI-based [165] network-booting [166] approach to provision instances. We used the Linux SCSI Target Framework (TGT) [167] for iSCSI-based network booting.

Ceph is an open-source storage platform that implements a highly reliable and scalable object storage on a distributed cluster. M2 uses Ceph's block storage interface for managing and snapshotting instance disk images.

For the Introspection Service, we extended M2 to support software introspection. This implementation maintains a database of software vulnerabilities populated using Canonical Ubuntu Security Notices [168][4]. Inherently, it uses IBM's open-source agentless crawler (IASC) [169] for scanning snapshots. IASC crawls through the filesystem tree present on a snapshot and generates frames corresponding to different OS and software package details. IASC stores the generated frames in a JavaScript Object Notation (JSON) [170] file. The vulnerability detection component of the Introspection Service then reads each frame and compares them against the blacklist present in the pre-populated database. After comparing each frame against the database, a list of vulnerabilities is generated and returned to the Orchestration Service. We are working on upstreaming the Introspection Service to M2. Note that our introspection implementation can be extended with other

[4]It is also possible to periodically query other open-source databases maintained by various non-profit organizations such as Mitre Corporation's Common Vulnerabilities and Exposures [144], National Institute of Standards and Technology's National Vulnerability Database [145], etc., and update the vulnerability database.

vulnerability detection services such as chkrootkit [35], OSCAP [171], or Linux Malware Detect (LMD) [36].

We also implemented our own Orchestration/Coordination Service, which coordinates among the services for introspection. It is implemented as a RESTful web-service [172]. Upon receiving an introspection request for a server, the Orchestration Service first creates a snapshot of the server's disk image, then maps the snapshot as a block device using Ceph's block device interface, followed by mounting the block device to a target directory. Now that the snapshot has been mounted for introspection, the Orchestration Service invokes the Introspection Service, passing the path to the target directory as an argument. After the Introspection Service returns the list of vulnerabilities, the Orchestration Service returns that list as the response to an introspection request. Before returning the list of vulnerabilities, the Orchestration Service also cleans up the state. i.e., it unmounts the mapped block device, unmaps the disk image snapshot, and deletes the snapshot.

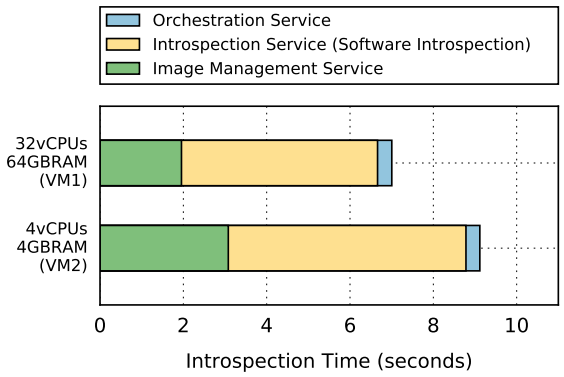


Figure 4.4: Dissection of time taken to process a single Software Introspection request across different NSI services considering two different VM configurations to host NSI.

4.3 Evaluation

In this section, we evaluate NSI. We first present the experimental setup. We then analyze NSI and its service's runtime under an increasing load. We then present the end-to-end introspection times of different vulnerability analysis mechanisms. Finally, we present the performance impact of introspection via NSI on workloads running on the introspected bare-metal servers.

4.3.1 Experimental Setup

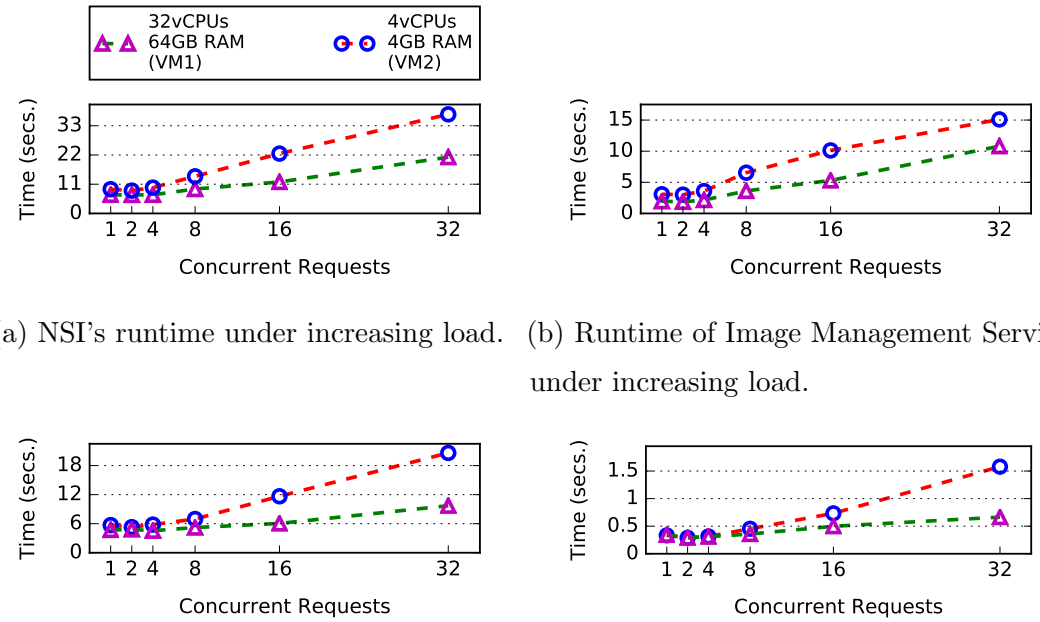
To evaluate NSI, on-demand bare-metal instances from the Massachusetts Open Cloud [173] were used. The bare-metal instances used during the evaluation have two 8-core Intel(R) Xeon(R) CPU E5-2650 v2 @2.60GHz (32 hyperthreaded cores), 64GB RAM and two dual-port 10 Gigabit Intel 82599 NIC's. All of the bare-metal servers to be introspected were running the RHEL 7.5 OS (provisioned from a 50GB base image). The servers had no local disks attached and the application data was stored on NFS drives mounted on the bare-metal servers which was not being introspected.

A three-node 98 TB Ceph storage cluster with 27 OSD's and 10 Gbit external and internal NIC's was used as the Image Management Service. Each experiment presented from here on is repeated five times, and the average of the five values is plotted. For all the experimental results presented in this section, none of the NSI components (responsible for introspection) were running on the introspected bare-metal instance. Therefore, the performances of applications running on the introspected bare-metal instances are not affected – irrespective of their compute intensities.

4.3.2 Runtime Analysis of NSI and its Components

In this section, we present an analysis of NSI and its components' runtimes. For these analyses, two different VM configurations are used to host the Provisioning Service, Image Management Service client, Introspection Service, and Orchestration Service. These VMs run CentOS 7.4. The first configuration (VM1) had 32 vCPUs and 64 GB RAM allocated, whereas the second configuration (VM2) had 4 vCPUs and 4 GB RAM allocated. No workloads were running on the bare-metal servers while they were being introspected. *Note that* the reason to conduct this experiment with two VM configurations is to provide insight to tenants operating at different scales. For example a tenant operating at a small scale should not worry about reserving many resources for introspection; whereas a tenant opreating at a larger scale should be able to estimate the the amount of resources required to introspect its massive infrastructure.

Fig. 4.4 presents a runtime dissection of the introspection of a single server across NSI services. As seen in the figure, in total, NSI requires from 7 to 9 seconds to process a single introspection request when deployed on VM1 and VM2, respectively. In both configurations, the introspection time is dominated by the Introspection and Image Management Service



(a) NSI's runtime under increasing load.

(b) Runtime of Image Management Service under increasing load.

(c) Runtime of Introspection Service under increasing load.

(d) Runtime of Orchestration Service under increasing load.

Figure 4.5: Runtime analysis of NSI and its services.

times, with the Orchestration Service taking the least amount of time.

Fig. 4.5 presents how the runtime of NSI and its services behave as the number of concurrent introspection requests issued to them is increased from 1 to 32. Fig. 4.5a shows the total introspection time of NSI under different numbers of concurrent introspection requests, whereas Figs. 4.5b, 4.5c, and 4.5d show the time consumed by the Image Management, Introspection, and Orchestration Services, respectively.

As seen in Fig. 4.5a, the time to process introspection requests is almost flat initially but starts to increase when trying to process more concurrent requests (8 in the case of VM1, and 4 in the case of VM2). As the number of concurrent requests increases, the runtime increase in the Image Management Service is more prominent. This increase is due to the increased network communication overhead between the Ceph client and the Ceph cluster. As seen in Fig. 4.5a, even for VM2 with 32 concurrent requests, the introspection time is less than 40 seconds, indicating that NSI can be deployed with modest resource requirements in production systems with a large number of servers that need to perform fast introspection.

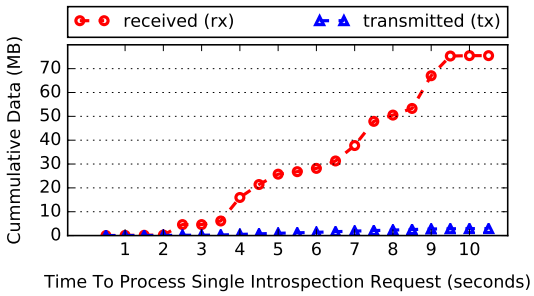


Figure 4.6: Network traffic between Ceph Client and Ceph Cluster when processing single introspection request.

Also, the runtime differences between VM1 and VM2 indicate that NSI can benefit from vertical scaling – especially when the number of expected concurrent requests increase.

Fig. 4.6 shows the cummulative network traffic between ceph client and ceph cluster when processing single introspection request. It was observed that while processing a single introspection request, ∼75 MBs and ∼3MBs data was received (rx) and transmitted (tx) respectively between the Ceph Client and Ceph Cluster.

Note that the communication between the Ceph client and the backend (i.e. the Image Management Service) causes the increase in end-to-end introspection time. The memory bandwidth is much higher than the network bandwidth and the blocks are fetched on-demand, thus memory bandwidth is not increasing the end-to-end introspection time. The Ceph client and backend are deployed with vanilla settings in our modest setup. Parameters for enabling read-ahead, minimum active clients, client-side caching, etc. were left unmodified/untouched. This can be one of the reasons why there was an increase of 5 seconds as the number of concurrent requests handled by the Image management Service increases from 16 to 32.

With the above results at hand, now let's consider an example for where we want to check for vulnerabilities in the kernel and installed software packages for a typical Borg cluster consisting of 10K bare-metal servers [155]. As shown in Fig. 4.5a, it takes ∼22 seconds to concurrently process 32 such requests with VM2, which indicates that a server with the same CPU-cores and memory can process ∼436 such requests every 5 minutes. Therefore, to introspect 10K servers every 5 minutes for basic vulnerabilities, 23 such servers should suffice. Furthermore, the network bandwidth usage to process each request (i.e., Fig. 4.6) shows that a 10 Gbps NIC should be sufficient for such an introspection server.

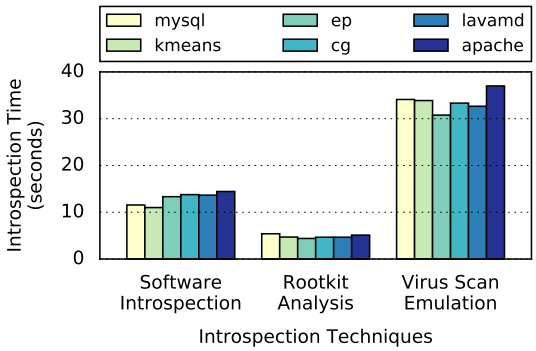


Figure 4.7: Software Introspection, Rootkit Analysis, and Virus Scan times while running various workloads on the introspected server.

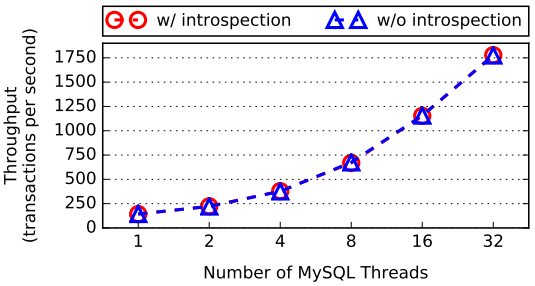


Figure 4.8: Impact of periodic introspection on MySQL workload performance with increasing compute intensity.

4.3.3 Different Introspection Mechanisms

Fig. 4.7 presents the Software Introspection, Rootkit Analysis, and Virus Scan introspection times with NSI while running various workloads on the introspected server. The basic Software Introspection scans for vulnerable OS and software packages on a server[5]. The Rootkit Analysis checks for the presence of rootkits using the `chkrootkit` [35] software tool. A rootkit is a software program that tries to gain unauthorized access to the system without being detected, The Virus Scan Emulation emulates the behavior of anti-virus software, i.e., to scan the contents of the entire filesystem. The content scan option of IBM's Agentless System Crawler [169] was used for this experiment.

In Fig. 4.7, the three introspection mechanisms were applied while the servers were

[5]NSI's Introspection Service (presented in Section 4.2.4) was used for detecting vulnerable OS and software packages

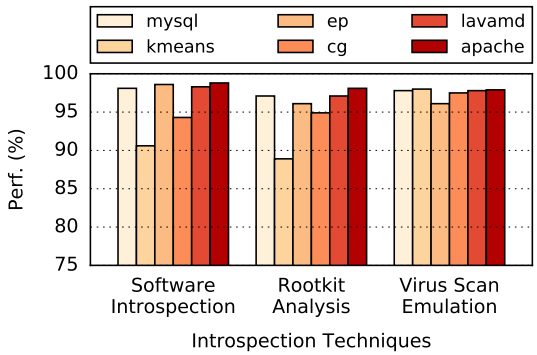


Figure 4.9: Impact of agent-based introspection on performance of applications running on the bare-metal server being introspected.

running six different applications: the MySQL [174] database server; the K-means clustering and LavaMD n-body simulation applications from the Rodinia benchmark suite [175]; the Apache web server [176]; and the Embarrassingly Parallel (EP) and Conjugate Gradient (CG) applications from the NASA Advanced Supercomputing Parallel Benchmark (NPB) suite [177] (class C). In this experiment, the applications were configured to use all of the 32 hyperthreaded cores available on the bare-metal server. The Sysbench [140] and ApacheBench [178] benchmarking tools were used to generate workloads for the MySQL database server and Apache web server, respectively. Both benchmarks were running on the introspected bare-metal instances during the experiments. NSI was running on a VM2 configuration.

As expected, the time taken to introspect a server with different introspection mechanisms varied a lot, since the different introspection mechanisms were performing significantly different analyses on the filesystem. As seen in Fig. 4.7, the time taken to introspect a server was similar across applications. The slight introspection time variance observed was due to the differences in installed software and filesystem content across different applications. *Note that* as the perturbations across different applications are minor, we can estimate the resources we need for the introspection with high confidence; if it varied by a lot for different workloads, such estimate would have been difficult.

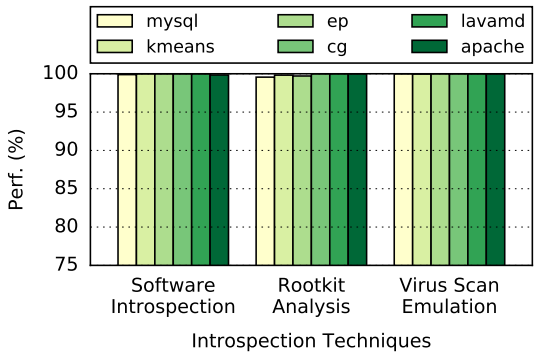


Figure 4.10: Impact of agentless introspection on performance of applications running on the bare-metal server being introspected.

4.3.4 Impact of NSI on Application Performance

We studied the impact of periodic agentless introspection on a single application when running with different application intensities and on different applications. In both cases, the server was introspected three times during the lifetime of each application and at an interval of one minute. Introspection of each application started after the bootstrap of that application, and application runtimes averaged around three minutes. VM2 configuration was used to host NSI services, and the applications and benchmarking tools used were the same as those presented in Section 4.3.3.

Fig. 4.8 presents the MySQL database server's OnLine Transaction Processing (OLTP) throughput (Transactions-Per-Second (TPS)) for two cases (with and without introspection) as the number of MySQL server threads were varied from 1 through 32. The first case (circles with dashed red line) presents the recorded TPS when the bare-metal server was not introspected and the second case (triangles with dashed blue line) shows the TPS when the bare-metal server was periodically introspected. As seen in the figure, the two lines match perfectly and there is no visible degradation in the OLTP throughput due to introspection.

Figs. 4.9 and 4.10 respectively present the impact of agent-based and agentless introspection on application performance. For both figures, the reported performance percentages are normalized against the application performance observed when there is no introspection. For Fig. 4.9, introspection mechanisms presented in Section 4.3.3 are installed as agents on the bare-metal servers and they are again configured to periodically introspect

the servers three times during the lifetime of each application and at one-minute intervals.

As seen in Fig. 4.9, agent-based introspection causes up to 12% performance degradation. On the other hand, as seen in Fig. 4.10 agentless introspection has negligible impact on the application performance.

Chapter 5

Bare-Metal Multiplexing

5.1 Introduction

Elasticity in the Cloud is supported with VMs or containers. Unfortunately, this is not sufficient in many use cases that more traditionally run on bare-metal systems. Many organizations still deploy different frameworks (e.g., OpenStack, Slurm, Spark, etc.) directly on bare-metal servers due to guaranteed performance and security advantages. Time-sharing of bare-metal servers across these deployments to extract the maximum value from the infrastructure is not considered due to concerns regarding security and privacy, and the complexity and cost of changing the ownership/access of bare-metal servers between frameworks. This leads to siloed bare-metal deployments and thus low aggregate resource efficiency at the organization-level.

The problem of low aggregate resource efficiency in data centers running heterogeneous workloads has been studied extensively [179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192] and the existing solutions can be broadly classified into three categories (based on the scheduling approach they support), namely *monolithic*, *two-level*, and *shared-state* [191]. Monolithic solutions utilize a single scheduler for making resource allocation decisions for all jobs [179, 180, 181, 188, 184]. As the number of frameworks operate under monolithic solutions increases, the complexity of the single scheduler increases as each framework will require its own metrics to be collected and parameters to be configured. Hence, the number of frameworks that can be supported with monolithic solutions is limited. Two-level solutions [183, 186] employ a meta-scheduling approach where a top-level scheduler offers/provides available resources to frameworks and each

framework makes its own scheduling decisions. Unfortunately, these solutions do not have knowledge about the resource utilization of the frameworks, and thus lack support for global optimization at the organization-level. Furthermore, since they generally rely on containerization technologies for resource allocation, they are susceptible to performance and security problems [193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206]. Shared-state solutions [191] expose global resource usage information to each framework so that they can acquire/release the resources using global concurrency control (in conjunction with policies for priority and starvation management). Like two-level solutions, they also lack support for global optimization at the organization-level. Furthermore, shared-state solutions may dictate modifications on the jobs/frameworks that can utilize them, or may only support specific workloads. None of these solutions (i.e. monolithic, two-level, or shared-state) lend the control of hardware configurations (e.g. controlling simultaneous multi-threading, etc.) to individual frameworks. Due to these reasons, they are not useful for breaking the bare-metal silos and enabling bare-metal time-sharing between frameworks.

We analyzed real-world specifications of secure bare-metal time-sharing in data centers and cloud platforms [147, 207, 208, 209], shortcomings and requirements of existing systems for resource time-sharing, and potential organization-level gains from bare-metal time-sharing (see §5.2). We list the set of critical requirements for a cross-framework bare-metal time-sharing meta-framework to be general and useful below:

- Time/Cost required to move bare-metal servers between the frameworks should be low. If these are too high, the gains from time-sharing will diminish §5.2.
- Frameworks should be secured and isolated from each other. This is especially important for security-sensitive organizations such as federal agencies, financial institutions, etc [147].
- Frameworks should have access and control over the hardware configurations (e.g., simultaneous multi-threading, virtualization, hardware-based security, power management, etc.) of the bare-metal servers they operate. As an example, frameworks running performance-sensitive workloads may want to turn off simultaneous multi-threading to prevent jitters, whereas other frameworks may want to keep it on to increase throughput.
- Global optimization to maximize organization-level metrics should be supported. This

is especially critical for organizations such as Google [180, 181], Alibaba [], or National Security Agency [210], where multiple groups operate multiple frameworks under one roof but the ultimate goal is to further organizational goals.

- Frameworks should not need significant modifications to their internal scheduling logic to benefit from meta-scheduling. Any such modification requirement would hamper adoption and limit generality. A good example to this is the difference between the adoption rate of Borg vs Mesos models. Even though the Borg model is more efficient, due to Borg's restrictions on how the frameworks have to define/execute jobs, Mesos model is more adopted.

To address the requirements of cross-framework bare-metal time-sharing and overcome the problems with the existing global scheduling and resource allocation solutions we propose. It enables global-optimization-aware time-sharing of bare-metal servers across co-located frameworks in a data center. It employs a hybrid of two-level and shared-state scheduling approaches. The top-level scheduler has access to the global resource utilization knowledge (similar to shared-state) and makes bare-metal resource (re)allocation decisions for the organization to optimize global metrics such as revenue. The second-level comprises of individual framework schedulers that are autonomous, isolated from the other frameworks, and do not have visibility into the global scheduling information. They only make the job placement decisions concerning the resources that are allocated to them (similar to two-level). This way, BareShala can run multiple frameworks without any modifications to their internal scheduling logic.

Bare-metal time-sharing is enabled by BareShala through a set of micro-services, namely: a network configuration service that isolates the frameworks from each other; a rapid provisioning service that installs software on servers using network mounted storage (in a security-sensitive environment, this service also performs bare-metal attestation); a bare-metal management service to perform out-of-band operations (e.g. power on/off, set boot device, etc.) on servers and maintain the bare-metal inventory; a telemetry service to gauge and manage the ongoing resource requirements of the frameworks; finally, an orchestration engine that performs server movement between the frameworks using these micro-services (based on the decisions made by the optimization model).

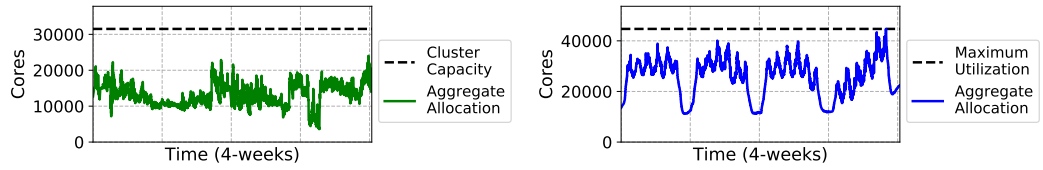
BareShala's approach of stateless provisioning of bare-metal servers and automated network isolation reduces the time/cost to move bare-metal servers between the frameworks,

enables the frameworks to execute directly on the hardware (without the requirement of a virtualization middleware), and control the hardware configurations of the bare-metal servers []. BareShala isolates the frameworks from each other by using VLAN/VXLAN technologies []. It uses trusted boot techniques [] (static root of trust and remote attestation) to detect server compromises. The top-level meta-scheduling decisions (that optimizes organization level metrics such as revenue, cost, and utilization) are made using the telemetry data collected from all the frameworks. The second-level decisions are made by the individual framework schedulers to optimize framework-level parameters (such as utilization, fairness, etc.). BareShala relies on the elasticity capabilities of the frameworks to dynamically add/remove servers.

Key contributions of this work are as follows: An architecture and prototype implementation of a global-optimization-aware cross-framework bare-metal time-sharing platform (BareShala), is presented. This platform enables organization-level optimizations without introducing additional complexities such as modifying existing frameworks/workloads, or requiring extensively complex schedulers, and supports workload elasticity while eliminating the performance and security concerns of virtualization/containerization; A trace-driven analysis of the potential benefits from time-sharing bare-metal servers between three frameworks is presented. Production traces from Microsoft Azure IaaS-cloud [211], Two Sigma Investments data analytics jobs processing financial data [212], and Los Alamos National Lab Mustang high performance computing [212] environments were used for this analysis. The analysis suggests an improvement of ∼XX% in aggregate CPU usage or a reduction of ∼YY% in server cost if time-sharing of bare-metal servers can be achieved with efficient bare-metal time-sharing; A performance evaluation of the prototype implementation using artificial traces.

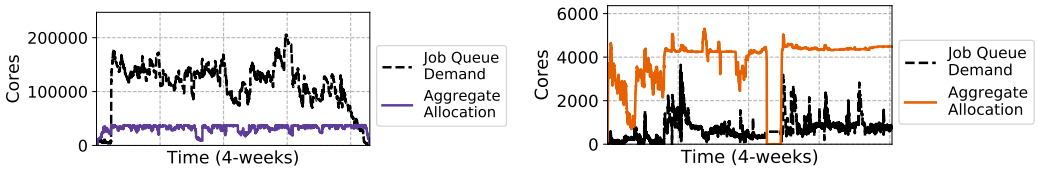
5.2 Trace-Driven Analysis

Organizations hosting multiple workloads in their data centers have realized that the utilization patterns between latency-sensitive and latency-tolerant workloads can be complimenting, and they have started exploiting these patterns to improve aggregate resource efficiency in their data centers [192, 183, 213, 190]. These solutions are based on the fine-grained time-sharing capabilities provided by virtual machines or containers. The rapid elasticity of virtual machine and containers enable organizations to alter the capacity of the



(a) Two Sigma Investments - Big Data Analytics.

(b) Microsoft Azure - Infrastructure-as-a-Service Cloud.



(c) LANL (Mustang) - High Performance Computing.

(d) CERN (ATLAS) - High Throughput Computing.

Figure 5.1: Resource usage patterns in real-world environments.

frameworks as and when their workload demand/utilization changes. This section presents a trace-driven analysis of potential gains from bare-metal multiplexing.

5.2.1 Framework Capacity vs Usage

The number of bare-metal servers allocated for a framework is typically determined via long-term capacity planning – while considering different metrics/constraints such as historical usage patterns, potential burst situations, allocated infrastructure budget, etc. For example, while determining the size of a framework that serves latency-sensitive workloads (e.g., cloud services, financial services, etc.), organizations tend to over-provision the bare-metal capacity to be able to support peak utilization demands, a potential surge in the number of users, or to be able to seamlessly handle failures. Moreover, it is well established that such workloads exhibit diurnal utilization patterns which can lead to low resource utilization for many hours in a given day. Such frameworks are critical to organizations (as they directly impact their revenue) and are often allocated more resources than their ongoing demand – thus resulting in low average resource usage. On the other hand, the size of a framework orchestrating latency-tolerant workloads (e.g., batch processing or machine learning training services etc.), is typically limited by budget constraints and as such they commonly observe high utilization and long job queues while waiting for resources to free

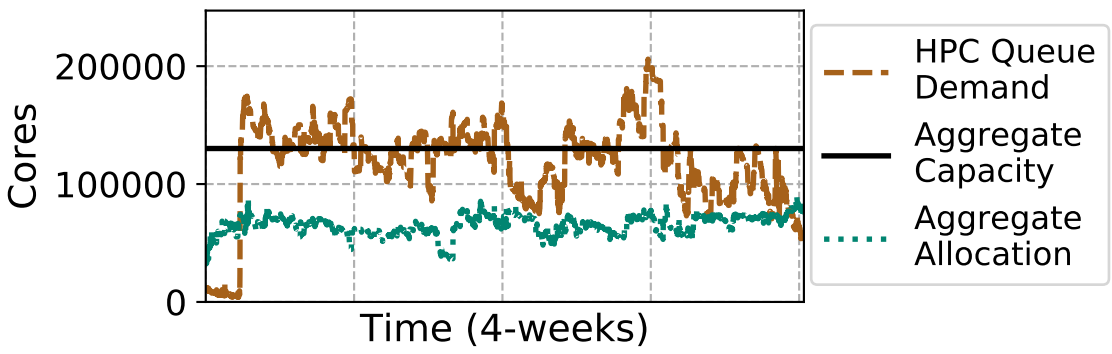


Figure 5.2: Bare-Metal Time-Sharing Potential.

up.

We looked at resource utilization traces from four production deployments running latency-sensitive (cloud and big data), and latency-tolerant (high performance computing and high throughput computing) workloads. Figure 5.1a shows the production trace for the data analytics jobs processing financial data at Two Sigma Investments [212] (31512 cores as total capacity). As can be seen in the figure, as the maximum utilization never reaches the total capacity even once during the entire period – indicating massive unused capacity present in the data center for weeks. In a private discussion with Two Sigma we also learnt that their data center is statically partitioned into multiple frameworks and despite this excess capacity in their data center, they frequently burst into public cloud for additional resources. Figure 5.1b presents the trace for short-lived virtual machines in Microsoft Azure Infrastructure-as-a-Service Cloud Platform (44712 cores as maximum aggregate utilization) experiences low resource utilization during weeknights and weekends [211] – observing several thousand core demand difference between working hours and weeknights. Figure 5.1c shows the production trace from Mustang that host high performance workload at Los Alamos National Lab [212] (38400 cores as total capacity). The resources utilization is near the actual capacity most of the times during the entire period. Moreover, there are jobs waiting in the queue that requires almost four times the actual allocated capacity. Finally, Figure 5.1d presents the production trace of U.S. ATLAS Northeast Tier-2 Center at Massachusetts Green High Performance Computing Center (MGHPCC) running single-node high throughput computing workloads originating from the ATLAS experiments at CERN [214] (5304 cores as maximum aggregate utilization).

5.2.2 Poetntial Bare-Metal Multiplexing Gains

Due to static framework sizing and varying resource demands, it is common to observe either resource *under-utilization* (e.g. frameworks that observe strong diurnal patterns in their resource demands) or resource *starvation* (e.g. frameworks that have steady high utilization) in individual frameworks – thus providing an opportunity to time-share underutilized bare-metal servers between co-located frameworks. This section presents a discussion on the potential efficiency and cost benefits that an organization can achieve by time-sharing unused bare-metal resources between co-located frameworks hosting heterogeneous workloads.

Trace Overview: To understand the potential benefits of bare-metal time-sharing, a trace-drive analysis of *four* real-world platforms presented in section 5.2.1 is performed. The workloads hosted by these frameworks are categorized as *On-Demand* (i.e., cloud and big data analytics) and *Batch* (i.e., high-performance computing and high throughput computing [1]). For the on-demand workloads, all the incoming requests are latency-sensitive and are required to be processed as soon as they arrive, whereas, for the batch workloads, incoming requests are latency-insensitive and are queued until resources are available for the job to be processed. All the workload traces except for the cloud are time-aligned. For the cloud workloads, neither the real timestamps for each incoming request nor the actual period of the trace was available. Next, for all the workloads except the cloud, the traces consist of all incoming requests. The cloud trace considered for this analysis consists of short-lived virtual machines that last for $\leqslant$ 3-days. Furthermore, for both big data and high-performance computing, the original capacity of the frameworks is used. Although the original capacity of the cloud framework was not available, its capacity was set to the maximum aggregate allocation observed in the entire trace plus a similar percentage of slack as seen in the bigdata analytics trace to handle bursty situations and failures.

Analysis Overiview: The analysis had the following characteristics: the total time simulated was 28-day and the time unit was minutes, i.e., a total of 40320 intervals; a *first-in-first-out scheduling policy* was used for all the frameworks to serve their incoming requests; all the requests for the on-demand frameworks were serviced without any delay - to ensure this, the maximum resource demand was populated depending on the time-sharing speeds; for the batch framework, only the requests that were submitted and completed

[1] Note that the high performing computing trace consisted of both single-server and multiple-server jobs.

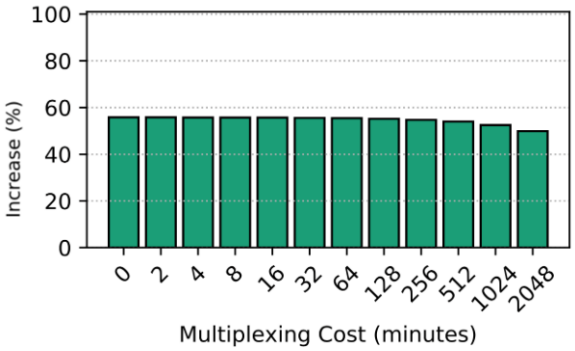


Figure 5.3: Observation 1

during the 30 days were considered; a batch request was only serviced *iff* sufficient compute resources were available to complete the request entirely.

To understand the importance of bare-metal time-sharing for organizations, the impact of three critical constraints related to bare-metal multiplexing were identified and studies. The variables are: (i) the speed at which bare-metal servers can be time-shared - the time-sharing cost was varied from 0 through 2048 minutes; (ii) the *fixed* or *variable* number of servers required to be reserved in on-demand frameworks to handle bursty situations and failures; and (iii) the impact of heterogeneous hardware to the workload performance - e.g., to understand the potential gains if the high-performance computing jobs were executed on the commodity hardware used in cloud frameworks.

In this analysis, we first calculate the unused bare-metal servers in the frameworks at each time interval under different constraints. Then, we measure the impact of providing the unused bare-metal server to the frameworks that can benefit from getting more servers, i.e., high-performance computing and high throughput computing in this case. To understand the impact of providing additional servers to these frameworks, we study three different optimization metrics. These metrics are: (i) improvement in aggregate resource efficiency of the unused infrastructure in an existing data center; (ii) potential cost savings when setting up a new data center; and (iii) reduction in queue times of batch jobs. Note that the trace-driven analysis presented in this section is conservative as it only analyses the benefits of time-sharing *unused* bare-metal servers. It does not try to preempt bare-metal servers that are occupied with workload jobs.

Key Observations: Here is a summary of the key observations First, when all the unused servers from the on-demand frameworks are made available to the batch frame-

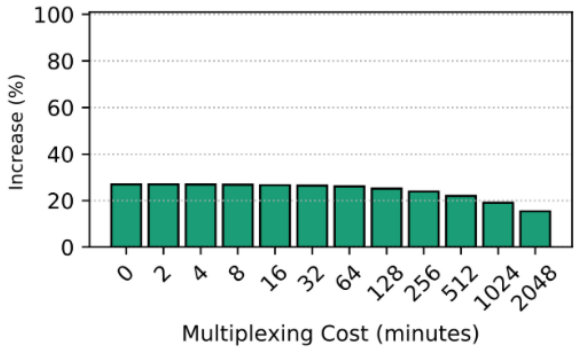


Figure 5.4: Observation 2

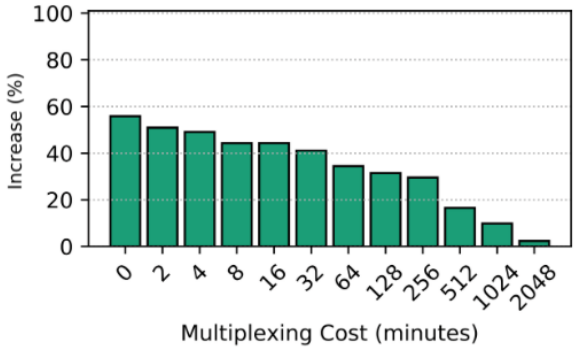


Figure 5.5: Observation 3

works, the batch frameworks can benefit from the extra resources despite high multiplexing overheads **??**. However, when reserving static amount of servers in (i.e. 25% of the total framework capacit)

It was observed that significant gains can be achieved even with high-overhead bare-metal multiplexing provided the *iff* no bare-metal were reserved in the on-demand frameworks to prevent (see Figure **??**) Static server reservations to support on-demand frameworks is bad for multiplexing The maximum servers required to be reserved is directly proportional to multiplexing speed - implying that significant gains can be achieved with fast time-sharing Significant gains can be achieved with fast time-sharing even with slower heterogeneous hardware

5.3 Architecture

Organizations can benefit from bare-metal time-sharing when hosting multiple frameworks with heterogeneous resource demand patterns in their data center 5.2. To enable

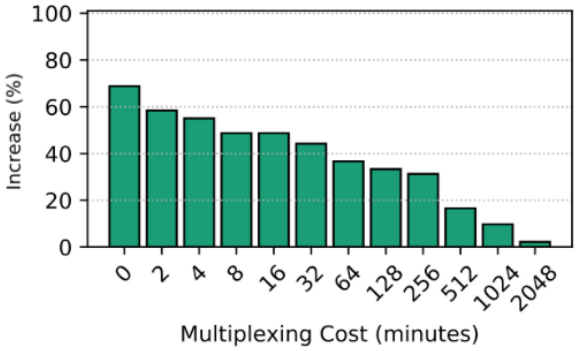


Figure 5.6: Observation 4

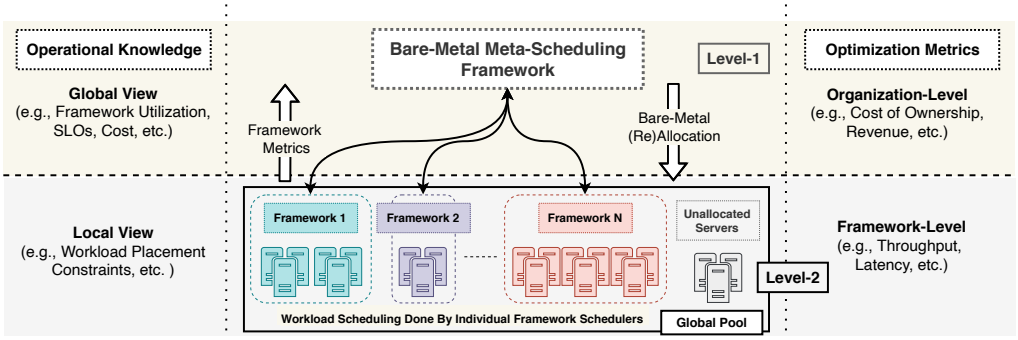


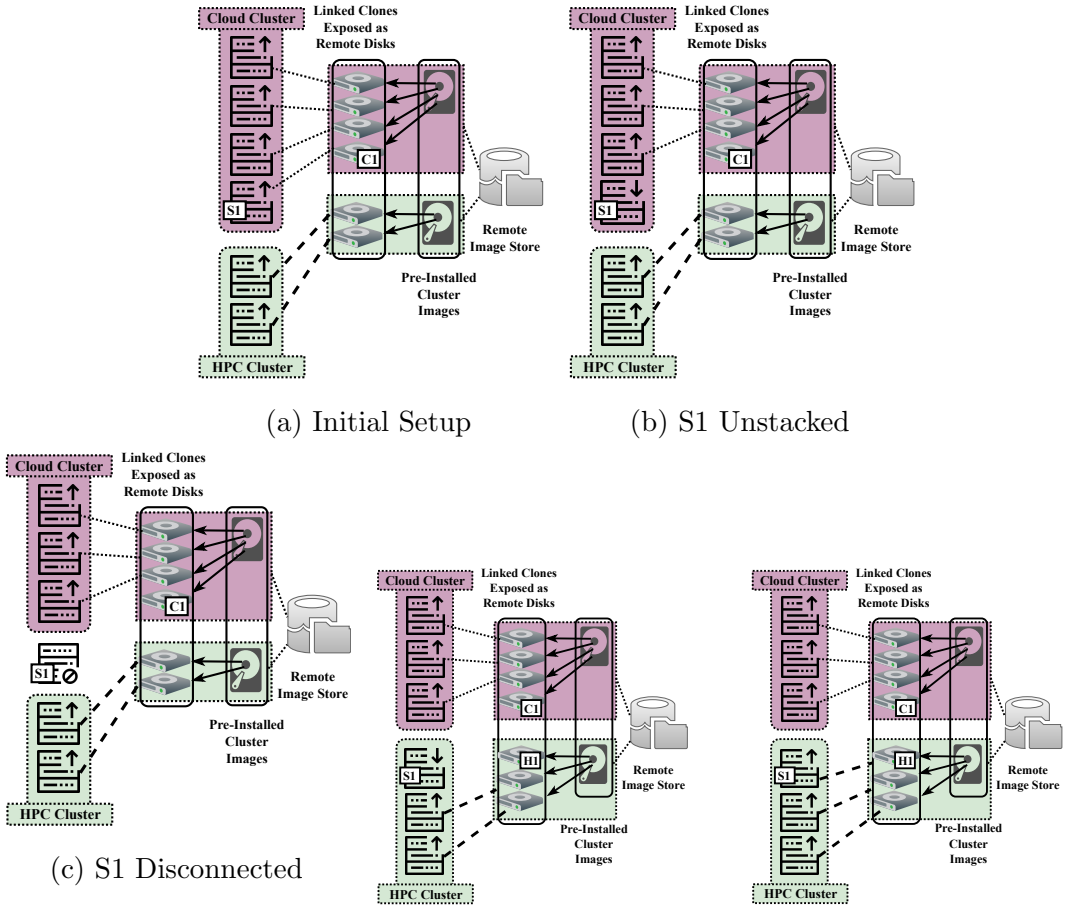
Figure 5.7: BareShala Overview

cross-framework bare-metal time-sharing in data centers, we propose BareShala.

5.3.1 High-Level Overview

BareShala employs a hybrid of two-level and shared-state scheduling approaches to enable cross-framework bare-metal time-sharing in data centers (see figure 5.7). The top-level scheduler has access to the global resource utilization knowledge via the telemetry data from each framework. It makes bare-metal resource (re)allocation/time-sharing/multiplexing decisions for the organization to optimize global metrics such as revenue, utilization, or throughput. The top-level scheduler identifies underutilized bare-metal servers allocated to a framework and re-allocates them to a framework that can benefit from extra resources. It incorporates high/low watermark-based thresholds to prevent bare-metal servers from being oscillated between frameworks.

The second level schedulers are the individual framework schedulers that are autonomous, isolated from the other frameworks, and do not have visibility into the global



(a) Initial Setup (b) S1 Unstacked

(c) S1 Disconnected

(d) S1 Connected (e) S1 Re-stacked

Figure 5.8: Server S1 re-allocated between frameworks.

scheduling information. They only make the job placement decisions concerning the allocated resources. This way, BareShala can run multiple frameworks without any modifications to their internal scheduling logic.

The frameworks and the workloads execute directly on the bare-metal servers, thus eliminating the performance and security problems of virtualization or containerization. They control the hardware configuration (e.g., processor virtualization, simultaneous multi-threading, trusted boot, secure execution, power management) of the bare-metal servers they possess through the APIs exposed by the baseboard management controller of the bare-metal server.

5.3.2 Bare-Metal Re-allocation

Each framework is treated as an isolated tenant and is secured from each other using network isolation techniques. All the frameworks managed by BareShala are isolated from each other at the network level (using VLANs or VXLANs), which means that the bare-metal servers in each framework share the same networks allocated to a framework but are not connected to other framework networks.

The frameworks support graceful removal and preemption of bare-metal servers associated. Each framework must expose an API that the top-level scheduler should invoke before removing the bare-metal servers from a framework to support the graceful removal of bare-metal servers. On the other hand, bare-metal server preemption is considered similar to a server failure in a framework, and the top-level scheduler removes the server without taking unique action.

Figure 5.8 presents an overview of how the bare-metal servers moved between the frameworks after the top-level identity which servers are to be reallocated. Bare-metal servers associated with a framework are provisioned in a stateless manner (i.e., the provisioned software state of the bare-metal servers resides on a disaggregated storage and is accessed over the network). Therefore, simply powering off the server de-provisions the software stack running on the server. Once the server is powered off, all the networks of the source framework connected to a server are detached. This is achieved by reprogramming the network switch ports to which the server's network interface cards are connected. Once the bare-metal server is completely removed from the source framework, it is connected a destination frameworks networks, followed by network booting from a pre-installed virtual disk containing the destination frameworks software stack.

5.3.3 Components

BareShala defines a set of micro-services to enable bare-metal time-sharing (see Figure 5.9), namely: network configuration service that isolates framework networks; stateless provisioning service that installs software on servers using network mounted storage (in a security-sensitive environment, this service also performs bare-metal attestation); resource management service to perform out-of-band operations on bare-metal servers and maintain the bare-metal inventory; telemetry service to gauge and manage the ongoing resource requirements of the frameworks; finally, an orchestration engine that performs

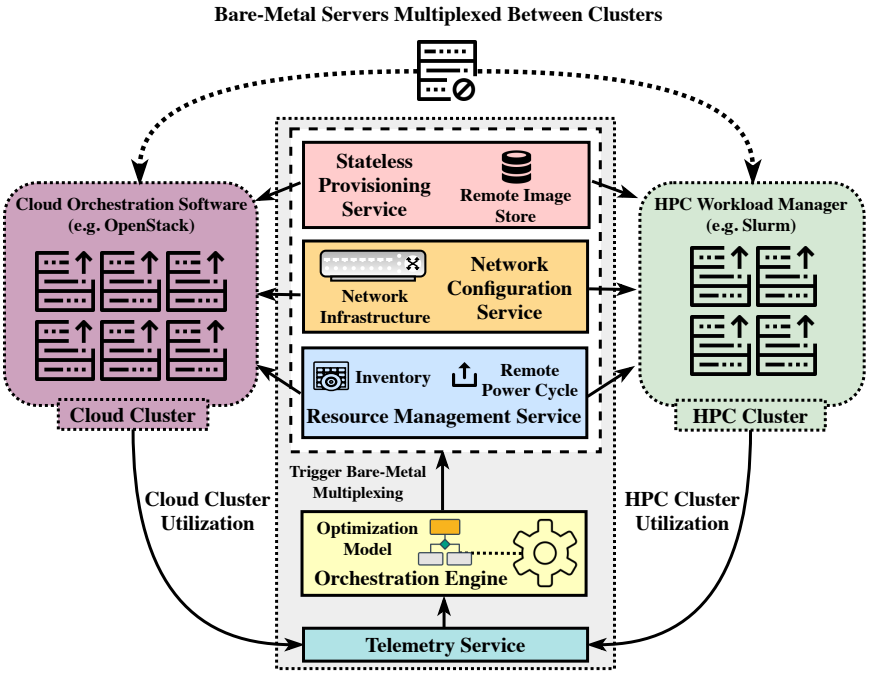


Figure 5.9: BareShala high-level architecture.

server movement between the frameworks using these micro-services (based on the decisions made by the optimization model).

Network Configuration Service: The Network Configuration Service exposes interfaces to program the networking infrastructure (i.e., network switch ports) in a data center to manage the network connectivity between the bare-metal servers. Furthermore, it enables BareShala to group the bare-metal servers in a framework and isolate them from other frameworks. Other networking features such as enabling Jumbo Frames or Spanning Tree Protocols can also be controlled using this service.

Stateless Provisioning Service: The Provisioning Service is the key to time-share bare-metal servers between frameworks. It helps in rapidly changing the personality of a bare-metal server based on its associated framework. It is used for (a) initial provisioning of a bare-metal server with the desired operating system and software stack pertaining to a framework, (b) saving the disk state of the bare-metal server when a server is released from a framework, and (c) re-storing a bare-metal server with the same disk state when it is re-assigned to a framework. Bare-metal servers are provisioned to a remote virtual disk residing on a highly reliable distributed storage. A one-time overhead is incurred to create an initial virtual disk containing the operating system, software stack, and initial configuration

scripts (for each framework). The Provisioning Service then leverages advanced features of the distributed file system to rapidly create copies (copy-on-write clones) of the initial virtual disk, followed by the network booting a bare-metal server from it. This service can also be responsible for verifying server integrity before it is provisioned for a framework (see Section **??**).

Telemetry Service: The Telemetry Service collects the utilization statistics (and including potential future resource demands) from each framework and exposes interfaces to share them with the Orchestration Engine. It maintains a database of server utilization statistics, which is periodically updated by the frameworks. Each framework records the ongoing usage of different server resources (e.g., memory, CPU, and network) to this database. Using these data points about frameworks and other constraints such as framework priorities exposed by the Telemetry Service, time-sharing decisions are by the Orchestration Engine.

Resource Management Service: The Resource Management Service is responsible for inventory management of the bare-metal server's data center resources and remote power cycling. In addition, this service is responsible for the bookkeeping of the data center resources (e.g., allocated bare-metal servers, VLANs) and exposes interfaces to query information regarding the same. Furthermore, it leverages the out-of-band server management capabilities in modern bare-metal servers to support the remote power cycle of the bare-metal server.

Orchestration Engine: The Orchestration Engine controls the workflow to time-share bare-metal servers among the deployed frameworks to improve aggregate resource efficiency. The organization deploying BareShala registers any essential constraints to be considered while making global optimization decisions (e.g., framework priorities). Based on these constraints and framework utilization and SLOs, the Orchestration Engine sequences the set of operations required to time-share bare-metal servers. The set of operations sequenced by the Orchestration Engine are driven by an optimization model implemented by the organization.

5.3.4 Prototype Implementation

Implementation: This section presents the implementation details of each component for the BareShala prototype used in this work. Each of the components are independent can be replaced with any specific implementation complying to the requirements, design

principles, and component description presented in the previous sections.

Hardware Isolation Layer (HIL): HIL [] is a multi-tenant network isolation system for bare-metal servers. HIL has a notion of projects to support multi-tenancy – where each project represents a tenant. The core functionalities HIL exposes are (i) allocation of physical servers to projects, (ii) allocation of networks to a project, and (iii) connecting allocated servers to a network. HIL maintains a database of available, and project-allocated bare-metal servers and networks. HIL can isolate bare-metal servers either using VLANs (OSI Layer-2) or VXLANs (OSI Layer-2 encapsulated in OSI Layer-4). HIL also provides tenant's with out-of-band server management capabilities such as remote reset, console access, etc. using *ipmitool* [].

BareShala leverages HIL to suffice it's requirement for (a) inventory management – using HIL's database where projects represents different frameworks, (b) remote power cycling, and (c) network isolation service. For the BareShala prototype presented in this work, OSI Layer-2 isolation mechanism (i.e. VLAN) was used. HIL acts as the Infrastructure Management Service for BareShala. The network operations (e.g. changing switch-port VLAN membership of a bare-metal server) HIL performed were asynchronous with no way to verify the if the it succeeded or not. For BareShala to perform bare-metal time-shareing it is required to verify if the switch-port VLAN membership of the intended bare-metal server is changed successfully before it becomes available for provisioning. For the BareShala prototype presented in this work, the functionality to verify network operation status performed by HIL was implemented and up-streamed it to HIL's source code.

Bare Metal Imaging (BMI): BMI [] is a multi-tenant bare-metal cloud service. BMI employs iSCSI-based network booting in order to achieve virtual-machine-like elasticity and agility for bare-metal server instances. It provisions bare-metal servers from a pre-installed cloud-image-like virtual disk residing on distributed remote storage. The core functionalities BMI exposes are (i) disk image creation, (ii) image clone and snapshot, (iii) image deletion, and (iv) server boot from a specified image. For the BareShala prototype presented in this work, BMI uses HIL's multi-tenancy capabilities.

BareShala leverages BMI to suffice it's requirement for a stateless provisioning. For the BareShala prototype presented in this work, Linux SCSI Target Framework (TGT) [] and Ceph distributed storage platform was used (NOTE: Reliable Autonomic Distributed Object Store Block Device a.k.a. RBD functionality of Ceph was used to manage the virtual disks). BMI acts as the Stateless Provisioning Service for BareShala. In order for bare-metal

time-shareing to work, it is required to preserve the existing disk state to for a server before releasing (de-provisioning) it from a framework. The default bare-metal de-provisioning operation in BMI deletes the remote boot disk. For the BareShala prototype presented in this work, the functionality to preserve disk state when de-provisioning and re-instantiate the server from an existing disk state when re-provisioning a server was implemented[2].

Telemetry and Orchestration Services: Resource utilization statistics (e.g. CPU usage, RAM usage, Network bandwidth consumption, etc.) are periodically collected using the sysstat [] Linux utility for every bare-metal server allocated to each framework are recorded to an NFS server. Each framework also periodically records (a) a value-proposition for bare-metal server it is willing to pay in order to acquire or maintain its access, (b) the number of bare-metal server it wishes to acquire, and (b) the number of bare-metal servers it is willing to release. A web-service runs in conjunction with the NFS server that provides API's to fetch all recorded information.

A python-based framework was developed that wraps around the aforementioned services and performs rapid bare-metal time-shareing to achieve global optimization. The Orchestration Engine periodically retrieves various framework-specific information from the API's provided by the telemetry service. Based on the information, the Orchestration Engine generates a list of servers to be time-shareed. The decision of which bare-metal servers need to be time-shareed is made by a policy implemented based on based on the optimization model (presented in Section XXX). Using HIL, the orchestration Engine first detaches the host framework's networks from the bare-metal server, power cycles the server, and then attaches the destination frameworks networks to it. Meanwhile, as the server is going through a Power-On Self-Test phase, the Orchestration Engine configures iSCSI targets using BMI – from which the bare-metal servers will be booted.

5.4 Evaluation

In our experiments and analyses we used OpenStack as our cloud management system and we used Slurm as our HPC management system. In Figure 5.10, we present the operation times in seconds while we are migrating nodes from an HPC deployment to a cloud deployment (HPC to OpenStack) and from a cloud deployment to an HPC deployment (OpenStack to HPC). For all migration scenarios, the amount of time spent

[2]We are in process to upstream this functionality to BMI

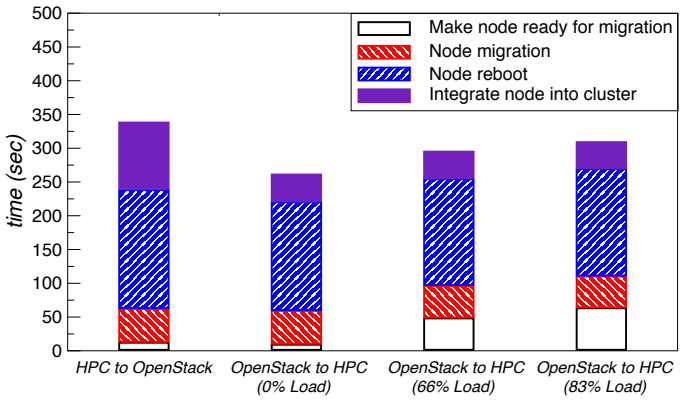


Figure 5.10: Node migrating times (seconds).

for node migration (operations associated with HIL) and node rebooting is the same. For the case of HPC to Cloud migration, the amount of time spent for making the node to be migrated ready is relatively low even though we assume that the HPC nodes are always running with high (above 90%) utilization. However, the amount of time spent for integrating a new node into the cloud management system is significantly high. All in all, migrating a node from HPC to cloud takes around five and a half minute. On the other hand, for the case of OpenStack to HPC migrations, we considered multiple cases where the node to be migrated had different loads.

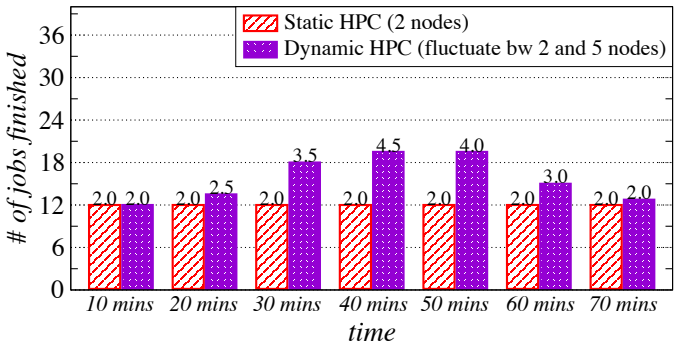


Figure 5.11: HPC throughput comparison of static HPC-Cloud clusters and dynamic HPC-Cloud with migration.

The total number of nodes in the cluster used for the following experiments is 11. In the static deployment of cloud and HPC, the HPC cluster has three nodes (including the Slurm controller node) and the OpenStack cluster has eight nodes one of which is the

OpenStack controller. We approximate a time-varying workload for the cloud by using a cosine pattern of demand on the OpenStack cluster. To approximate the cosine curve, we sampled it at ten minute intervals and stopped or started some of the VMs running on the cloud based on the sampled value. Due to the variance generated by the time-varying workload on the OpenStack, the number of worker nodes on the HPC cluster in the dynamic deployment varies between two and five.

In Figure 5.11, we compare the throughput of a static HPC deployment with a dynamic HPC system that shares nodes with an OpenStack deployment. The number of nodes on the static deployment is fixed to two, whereas, based on the load on the OpenStack deployment the number of nodes in the dynamic deployment vary. The observed throughput values are in line with the amount of load OpenStack cloud observes.

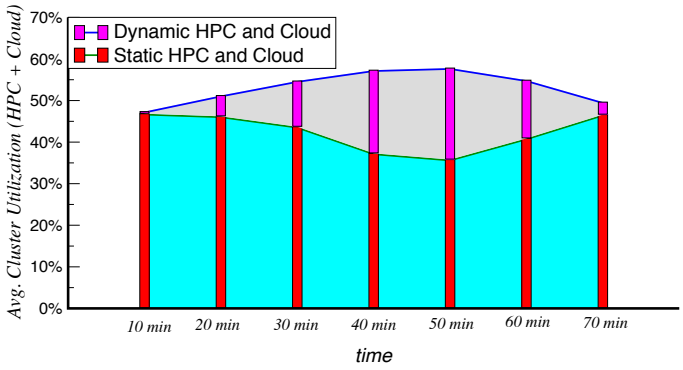


Figure 5.12: Utilization comparison of static HPC-Cloud clusters and dynamic HPC-Cloud with migration.

Finally in Figure 5.12 we present the overall utilization of the whole cluster supporting both the HPC and Cloud systems. As seen from the figure, the overall utilization of the dynamic deployment is comparably higher compared to static deployment.

Chapter 6

Job Co-scheduling in Batch Systems

In both high performance computing (HPC) and the Cloud, it has long been recognized that isolating the jobs of different users by running on distinct CPU cores is inefficient. There are times when a CPU core lies idle, and yet no other job can consume those unused CPU cycles. The operating system is designed to allow distinct processes to share the CPU cycles of a core. But administrators prefer to isolate batch jobs on distinct cores, so as to create a transparent and fair chargeback system. The sum of the number of cores declared by all current jobs must add to at most the number of existing cores.

A new approach is proposed that is insensitive to operating system optimizations. The active job set is dynamically adjusted at runtime (after job submission) to decide whether to over-commit CPU cores, under-commit or turn on hyper-threading (SMT). Incompatible job sets that pollute the cache or cause thrashing are automatically avoided. Performance increases by as much as 75% are seen in some instances, while still providing soft guarantees that limit the degradation of any individual job as compared to its ideal performance when run standalone on that same node. This mitigates the classical administrator's dilemma — system throughput or fairness in the chargeback.

6.1 Introduction

Batch-style many-core workloads for single-process jobs are the future. Such jobs appear both in traditional HPC and in the Cloud (e.g., through IaaS). Yet application

writers are notoriously poor at predicting how many CPU cores will be needed, and they often declare a requirement for more cores than they need. This happens especially when a job execution may pass through several phases, with different resource requirements in each phase. This results in wasted CPU cycles.

Existing work in this area concentrates on predicting performance for a known workload with known characteristics in order to decide questions of co-scheduling (during batch job submission for HPC clusters [77, 76, 78]) or co-location (for corporate datacenters, to co-locate known base applications with background "best-effort" jobs [79, 80, 81, 82, 83, 84, 85, 86, 87]). But both large HPC centers and the Cloud are often forced to process diverse workloads whose characteristics are not known in advance, In the real-world case of diverse workloads, previous systems cannot recover when they select a bad set of actively running jobs.

Instead, this work dynamically selects the active job set at runtime. It uses checkpoint-restart to rapidly save the state of any running jobs that are discovered to hurt the performance of other jobs that have been co-scheduled/co-located. Thus, no work is wasted (i.e., no CPU cycles are "lost").

This approach introduces two critical issues. The second, especially, has not been addressed satisfactorily in the literature.

1. how to determine whether the performance of a particular job is less than its ideal performance (as the sole application on that node); and

2. what eviction or replacement policy to employ when the active job set is found to yield less than ideal performance.

The first issue is answered by running each new application in the batch queue for a short amount of time as the sole application on that computer, and then measuring its performance, considered as the *ideal throughput*. When the ratio of current to ideal throughput degrades below a certain level, one of the jobs can be checkpointed and evicted. This ensures a *soft guarantee* of limited degradation. In long-running jobs, the ideal throughput would be periodically re-measured to account for different base workloads as the job passes through separate computation phases.

The second issue of an ejection or replacement policy when the soft guarantee is violated is addressed through an AIFD algorithm. AIFD is inspired by the Additive

Increase/Multiplicative Decrease (AIMD) algorithm for TCP congestion control [215]. AIFD (Additive Increment/Full Decrease) is described in detail in Sections 6.2 and 6.3. Conceptually, it dynamically adjusts the currently active workload to choose the optimal level of over-commitment. The four over-commitment levels or phases are:

A. *under-commit:* under-commit ratio of threads to cores

B. *over-commit:* over-commit ratio of threads to cores

C. *hyper-threading:* enabling hyper-threading (while fixing the current number of scheduled threads used in phase B)

D. *hyper-threading + over-commit:* further over-commitment beyond phase C

Contributions: This work makes the following contributions:

1. *Dynamic reactivity:* An **Additive Increment/Full Decrease (AIFD)** strategy dynamically measures the current throughput against a measured ideal throughput. The level of over-commitment is adjusted accordingly.

2. *Dynamic hyper-threading:* The system administrator need not worry whether to configure a computer to turn hyper-threading on or off, since the system dynamically re-tunes itself at runtime.

3. *Flexible re-scheduling:* No work is lost when removing a job from the active job set, due to checkpointing [216].

Two particular successes in the experimental evaluation are noted here. Section 6.4.5 shows that when jobs can migrate between two computers running two instances of AIFD, the instances re-organize themselves to favor CPU-intensive or mixed CPU-/RAM-intensive jobs. Section 6.4.6 compares AIFD against the well-known Slurm resource manager for eleven real-world applications drawn from four HPC benchmark suites, and shows AIFD out-performs standard Slurm by 75%.

6.2 Overview

We assume a many-core computer running under an Additive Increment/Full Decrease (AIFD) model. This model treats the execution of a job through alternating phases

of *production epochs* and short *re-balancing periods*. A *production epoch* is a fixed period of time during which jobs are executed. At the end of execution, the re-balancing period AIFD: (i) acquires measurements of the performance (throughput) of the jobs that just executed; and (ii) makes decisions whether to start/re-start some jobs from the batch pool, stop (checkpoint) some currently running jobs, switch from one phase of AIFD to a different phase, and so on.

The details of the decisions made in the re-balancing period are what distinguish one AIFD phase from another. In particular, when throughput is measured in a re-balancing period, the actual throughput of the workload for that epoch is measured, and compared to an ideal throughput. A tunable threshold ratio of actual to ideal throughput determines if performance during the last epoch is considered good or bad. A "good" performance indicates that one may add another job from the batch pool into the currently running job mix. A "bad" performance indicates that one may "step back" to a previous job mix, or change to a different phase, or invoke a full decrease (i.e., begin again with an entirely new job mix chosen at random from the batch pool).

Next, we present a meta-algorithm, Algorithm 1, which, in principle, can describe any of the four phases. The only difference in each of the phases is the policy chosen during re-balancing.

6.2.1 AIFD: A System in Four Phases

We next consider in greater detail both startup of an AIFD-based batch queue and execution of each of its four phases. Note that the system may transition among phases according to Figure 6.1.

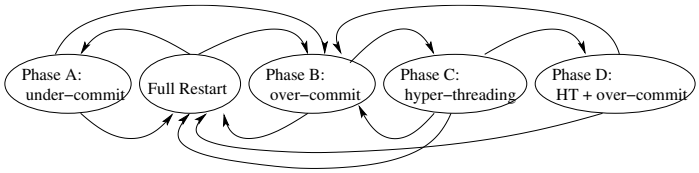


Figure 6.1: The four phases of AIFD ("Full Decrease" corresponds here to "Full Restart".)

If the system is beginning, or has just completed a "Full Decrease" as part of AIFD, then a random subset of available jobs from the batch pool is chosen. Recall that the batch pool consists of checkpoint image files, and we are either executing new jobs or restarting

Algorithm 1 Meta-algorithm (core algorithm for each phase):
different policies produce different variations

```
while True do
    Initialize and measure ideal performance for any
      recently queued job(s) and place in the batch-pool
    Execute a specific policy based on
      current system state
    Start new production epoch
    while not end of production epoch do
        Periodically check for completed job(s)
        if any job(s) completed then
            Replace from pool of jobs
    # Production epoch finished; begin re-balancing
    Get performance of all running job(s)
    Process end of production epoch (re-balancing)
    Update system state (e.g., new phase)
```

previously executing jobs.

AIFD departs from traditional batch queues in allowing for a variable level of over-commitment and a variable combination of jobs. After an initial production epoch, one must decide whether to attempt to raise the throughput, or to fall back to a different combination of job (full decrease). If the ratio of actual throughput to ideal throughput over the last epoch is above a tunable threshold, then the over-commitment level is raised (an additional job is launched/restarted). If the ratio of actual to ideal falls below the desired threshold, then either one reverts to the previous epoch, or else a full decrease is incurred in the AIFD scheme. (See Figure 6.1.)

During a production epoch, each of the co-scheduled jobs is run toward the end of the production epoch (or toward completion if it completes during the epoch). A CPU performance counter for cumulative CPU instructions is measured both at the beginning and the end of a production epoch. (This is the only low-level hardware information used.) This allows one to determine the *actual throughput*, as opposed to the *ideal throughput* incurred if a job were to run in isolation.

Next, we consider the policies for the four AIFD phases.

Full Decrease At the beginning of AIFD and in certain cases during the execution of AIFD, there is no prior indication of what might be a set of compatible jobs to be co-scheduled. In such cases, a random set of jobs from the "batch pool" is selected as a new job mix. The total number of threads in the job mix must be less than or equal to the total number of CPU cores. (Informally, this is referred to as an over-commit ratio of 1.0.) After a full decrease, after the first production epoch at an over-commit of 1.0, AIFD will attempt to run in under-commit mode by removing one job at random from the current job mix. If the performance actual throughput (absolute throughput, irrespective of the ideal throughput) improves, then the system enters Phase A (under-commit). Otherwise, the job mix at over-commit of 1.0 is restored, and the system enters Phase B (over-commit).

Phase A (under-commit) Phase A is concerned with the case of an over-commit level less than 1.0. The number of executing threads is less than the number of CPU cores. (See Figure 6.1.) The details of under-commit are described later in Section 6.3.6.

Phase B (over-commit) The concept of a degradation ratio plays an important role in phases B and D, where phase D refers to hyper-threading plus over-commit. During a re-balancing period, the total actual throughput of all co-scheduled jobs combined is compared against the total ideal throughput of all of the co-scheduled jobs. The result of this comparison determines either an increase or a decrease for the next production epoch. The ratio of the actual throughput to the ideal throughput determines a *degradation ratio* during the latest production epoch.

If the degradation ratio is larger than the configured degradation threshold (implying a small degradation), then an additive increment is chosen for the next production epoch, based on adding a new job from the batch pool at random. (Note that degradation ratios are less than one, and a degradation ratio of 1.0 implies no degradation from the ideal throughput. Hence, larger degradation ratio implies smaller degradation.) The action taken during an additive increase is described later.

If the degradation ratio is smaller than the previously chosen degradation ratio (implying a large degradation), then a decrease is chosen for the next production epoch. The action taken during a decrease may consist of a "step back" or a "full decrease". See Section 6.3.5 for details.

Note that a given job mix will typically reach a "plateau" in which the over-commit level oscillates by attempting to add a random job, and then falling back to the previous job mix.

Phase C (hyper-threading: HT) When, in Phase B, it is no longer possible to increase the level of over-commit, then an attempt is made to enable hyper-threading. Hyper-threading is turned on for the next production epoch. If the degradation ratio improves, then there is a transition from Phase C to Phase D (HT+over-commit). If the degradation ratio becomes worse, then the system reverts to Phase B. (Note that a standard technique of using Linux's CPU core affinities is employed to dynamically enable or disable hyper-threading. The computer is configured to enable hyper-threading, and the DMTCP checkpointing package provides a plugin that can selectively set the core affinities of a process to the first half of the virtual cores (i.e., the physical cores), or to all virtual cores.)

Phase D (HT+over-commit) After entering Phase C (hyper-threading: HT), the system immediately enters Phase D at the next production epoch. On each successive production epoch, if the degradation ratio is larger than the configured threshold, then an additive increase is triggered by adding a new job from the batch pool at random, similarly to Phase B. Similarly to Phase B, if the degradation is worse, than a "step down" (removal of one job), or else a "full decrease", is triggered. See Section 6.3.5 for details.

6.3 Degradation Thresholds and the Four AIFD Phases

6.3.1 Calculation of Degradation Ratios

When computing the ratio for many currently running jobs, we calculate the total actual CPU instructions executed (across all jobs) during one production epoch, and then the total ideal CPU instructions that would be executed (across all jobs, in one epoch). The ideal CPU instructions are the total of CPU instructions that would be measured for each job running in isolation. In principle, a weighted average could be used to distinguish RAM-intensive from CPU-intensive jobs, but this was not found to significantly change the results.

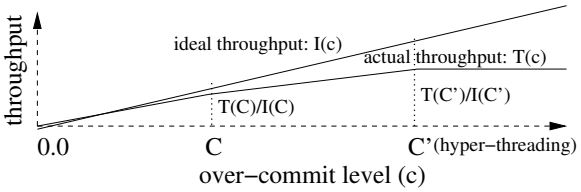


Figure 6.2: Actual throughput versus ideal throughput; assume C is the number of CPU cores, and that a switch to hyper-threading occurs when there are C' threads

6.3.2 Degradation Threshold Ratios for Phases B and D

In this section, the degradation threshold ratios to determine whether to further over-commit are described here in overview. The following Section 6.3.3 derives the actual formulas used for the thresholds.

For simplicity, let us first consider the homogeneous case, in which all jobs in the workload have identical characteristics. This would be typical of a user at an HPC site carrying out a parameter sweep in which many runs of a simulation are run, each job having different input values.

We fix the following notation, to be used throughout the rest of the work. The parameters i, t_1, t_2 and t_3 below are the constant parameters in our linearized model.

C	:	number of cores
C'	:	number of threads after which throughput overly degrades
Δ	:	change in number of threads
c	:	$C + \Delta$ (current number of threads)
D	:	base degradation threshold ratio (0.7 in Sect. 6.4)
$I(c)$	:	ideal throughput
$T(c)$	:	actual throughput
$T(c)/I(c)$	:	degradation of job mix

The idealized (optimal) throughput of a process will increase linearly with the number of threads as $I(c) = ic$.

Let us further assume for simplicity that each job has only a single thread. Continuing in this vein, we approximate the ideal throughput as $I(c) = ic$, and the actual

throughput as piecewise linear:

$$\begin{aligned} T(c) &= t_1 c && [\text{for } c <= C] \\ &= t_1 C + t_2(c - C) && [\text{for } C < c <= C'] \\ &= t_1 C + t_2(C' - C) + t_3(c - C') && [\text{for } c > C'] \end{aligned}$$

In Section 6.3.3, we will derive a model for deciding when to switch modes — to stop over-committing, and to test out hyper-threading. A *degradation ratio threshold* $\deg(c)$ will be specified in advance as part of the model. The AIFD scheduler will successively add a new job (and hence, increment the over-commit level) until the actual degradation ratio, $T(c)/I(c)$ becomes lower (i.e., worse) than the degradation ratio threshold. Hence, we will stop the normal over-committing and begin to use hyper-threading when there are c threads, and:

$$T(c)/I(c) = T(c)/(iC) < \deg(c)$$

The value $T(c)$ will be based on an actual measurement of the total throughput of the current jobs during the last epoch of computation.

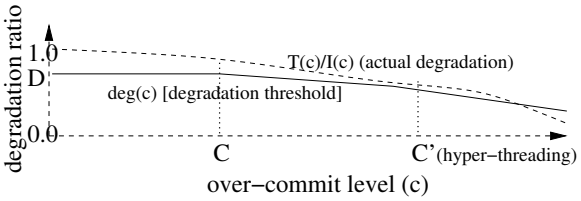


Figure 6.3: Limits of Phase D (HT + over-commit)

We denote by C' the number of threads present when we stop over-committing. The parameter C' will be found dynamically by determining when

$$T(c)/I(c) = T(c)/(iC) < \deg(c).$$

Our choice of the degradation ratio function, $\deg(c)$, will depend on D, t_2/t_1, and t_2/t_1. These three parameters will be fixed and assigned in advance. This process of over-committing until the actual degradation ratio falls below the degradation ratio threshold is summarized in Figure 6.3.

6.3.3 Phases B and D: When to Over-commit

Assuming that one has a mechanism for measuring both ideal throughput and actual throughput during a given epoch of computation, the next remaining question is under what circumstances to move on to the next level of over-commit.

Recall the notation from the beginning of Section 6.3.2. We wish to derive a simple, semi-empirical formula for deciding the threshold at which the system will accept to over-commit by another job, or to switch to hyper-threading. Hence, we will assume $I(c) = ic$ and $T(c) = T(C + \Delta) = t_1 C + t_2 \Delta$, for constant parameters t_1 and t_2, where $0 < t_2/t_1 < 1$ and t_2/t_1 represents the expected difference in throughput when one more thread is added beyond the initial C threads.

Phase B: Degradation Threshold We wish to derive $\deg(c)$, the threshold at which to accept another job is $T(c)/I(c) > \deg(c)$. Recalling that $T(C + \Delta) = t_1 C + t_2 \Delta$, we have:

$$\begin{aligned}
\deg(c) = \deg(C+\Delta) &= \frac{T(C+\Delta)}{I(C+\Delta)} \\
&= \frac{T(C)}{iC} \cdot \frac{iC}{i(C+\Delta)} \cdot \frac{T(C+\Delta)}{T(C)} \\
&= D \cdot \frac{C}{C+\Delta} \cdot \left(1 + \frac{t_2 \Delta}{t_1 C}\right) \\
&= D \cdot \frac{C + t'\Delta}{C+\Delta}, \\
&\text{where } t' = t_2/t_1 \text{ and } 0 < t' < 1 \\
&\quad \text{and } T(C+\Delta) = t_1 C + t_2 \Delta
\end{aligned}$$

Hence, in the implementation, if $D = T(C)/I(C)$ is the base degradation ratio, then the minimum degradation ratio required in order to over-commit by one additional job is $\deg(C + \Delta) = D \cdot (C + t'\Delta)/(C + \Delta)$.

Phase D: Degradation Threshold In a similar manner, once the maximum over-commit is reached and a further over-commit is rejected, one tries to turn on hyper-threading and continue to over-commit additional jobs while hyper-threading is turned on. Let $D' = \deg(C + \Delta')$ be the degradation ratio when hyper-threading is turned on. (That is, hyper-threading was turned on when Δ' additional threads had been over-committed.) The

condition for over-committing further during hyper-threading is similarly:

$$\deg(C' + \Delta) = D' \cdot \frac{C' + t''\Delta}{C' + \Delta},$$
$$\text{where } D' = \deg(C') \text{ and } 0 < t'' < t' < 1$$

and C' is the number of threads that were active when hyper-threading was turned on, and D' was similarly the degradation ratio in effect at that time.

6.3.4 Replacement Policy for Jobs that Complete

If a job completes during a production epoch, then a replacement is required. There are two cases. First, if a job completes during the last interval of a production epoch, then there is no replacement. Second (if the job completes before the last interval), the job is replaced by another job from the batch pool, subject to one constraint. When the current production epoch began, the threads of any job that was added to the job mix were not allowed to cause AIFD to exceed the over-commitment level for that epoch. Similarly, the replacement job must also not cause AIFD to exceed that over-commitment level.

6.3.5 Transitions between AIFD Phases

The implementation details for when transitions between phases (Figure 6.1) are omitted due to lack of space. The primary principles when the current degradation threshold ratio is not satisfied are: (a) attempt to revert to the job mix of the previous production epoch if possible; (b) *second chance policy* (replace an additional job from the batch pool); and (c) full decrease (revert to "full restart" state).

6.3.6 Phase A: The Under-commit Policy: A Special Case

We enter Phase A if after a full restart we observe performance degradation for the running jobs. In this case, we continue to under-commit so long as the actual throughput increases in absolute terms. (This typically indicates that there was thrashing.) Once the absolute throughput decreases, we revert to the previous production epoch, and continue with that job mix at a "plateau".

Eventually, a job will complete and be replaced. In this case, we start adding an additional job at each production epoch so long as the throughput increases. When that fails, we revert to the previous production epoch, but now at a second "higher" plateau. If

a job completes at this second plateau, then a full restart is invoked, rather than attempt further replacements.

until we keep observing an increase in the absolute throughput. Once the absolute throughput of the system stops increasing, i.e., when we hit a plateau, we keep running in the plateau mode until some job(s) completes thus resulting in a replacement. Upon replacement we start adding more jobs in the under commit phase as the equilibrium might have been disturbed due to the replacement. We keep adding jobs while running under-commit phase until we hit the second plateau or the number of running threads equals the number of physical cores available on the node. If a replacement is observed while running in the second plateau mode during the under-commit phase, we do a full restart as the equilibrium was broken and it might be costly to try and repair the system any further.

6.4 Experimental Evaluation

We evaluate our implementation of AIFD algorithm against multiple configurations of Slurm with EASY backfill algorithm. We first verify the proposed algorithm using a set of synthetic benchmarks that we developed. The synthetic benchmarks cover the case of homogeneous (identical) single-threaded jobs (Section 6.4.2), the case of heterogeneous single-threaded jobs (Section 6.4.3), and the case of multi-threaded jobs (Section 6.4.4).

Sections 6.4.5 describes how AIFD re-balances in a multi-node setup while and 6.4.6 presents the performance of AIFD with real-world applications.

Section 6.4.7 shows that for a fixed workload of jobs for AIFD, one can vary the AIFD base degradation ratio from 0.5 to 0.8, and all AIFD runtimes are within 10% of each other. Hence, although a ratio of 0.7 was used for these experiments, one would not expect to have to re-tune for different computer hardware.

Next, in interpreting the graphs of this section, note that where there are multiple instances of AIFD vertically positioned above each other, they represent distinctly chosen random job mixes. In the case of Slurm, even for a fixed job mix, the order in which the jobs are presented to Slurm will affect its running time. Where there are multiple instances of Slurm vertically positioned above each other, they represent distinct orderings of a single job mix (e.g., in Figure 6.12).

System	CPU	L3 Cache	Memory	Location
Sys-1	Intel Xeon E5-2650, dual-CPU, 28 HT cores	35 MB/CPU	256 GB	MGHPCC
Sys-2	Intel Xeon E5-2650, dual-CPU, 24 HT cores	25 MB/CPU	128 GB	MOC
Sys-3	AMD Opteron 8346 HE, 16 non-HT cores	2 MB shared	128 GB	N.U.

Table 6.1: Systems used for evaluation

6.4.1 Batch Queues: Experimental Design and Setup

We used three different systems for our experiments. Table 6.1 shows a summary of the configurations of the three systems. Each experiment was repeated three times for AIFD, and the jobs were presented to Slurm and AIFD in the same randomly generated sequence for each run. All results with AIFD include the checkpoint and restart overhead. We use DMTCP release version 2.5.0 for checkpoint-restart.

Slurm configuration We use Slurm release version 17.02 configured with the EASY backfilling scheduling algorithm.

AIFD configuration Sections 6.4.2 to 6.4.4 demonstrate the efficacy of the AIFD algorithm using a synthetic benchmark. Unless stated otherwise, Sys-1 was used for the experiments in sections 6.4.2 to 6.4.4 and 6.4.6. The synthetic benchmark implements an iterative algorithm, where each iteration touches the contents of a large, pre-allocated array. We also use this to validate the model (presented in Section 6.3.2) and to present the intuition about the model.

Note that the AIFD algorithm can potentially defer the execution of jobs that don't work well with each other, which can lead to starvation for such jobs. To mitigate this issue, we use an aging strategy based on exponential decay for all the experiments. This ensures that the scheduling is fair and that no jobs are delayed indefinitely.

We run AIFD with a base degradation ratio of 0.7, which means that we accept a degradation of 70% over ideal throughput in the normal case. An epoch length of 50 seconds was a used for all the experiments.

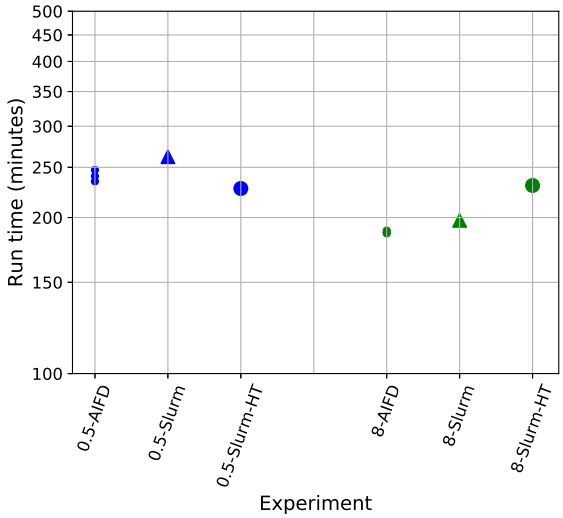


Figure 6.4: Homogeneous workload with synthetic benchmarks. Results are shown for two different cases: when the working set size of the benchmark is 0.5 MB, and when the working set size of the benchmark is 8 MB (Running time: AIFD, Slurm, and Slurm-HT for different workloads with synthetic benchmarks).

6.4.2 Synthetic benchmarks: Homogeneous case

Our aim here is to investigate the efficacy of the AIFD algorithm in the extreme case when the jobs in the system are homogeneous. This case is often seen in an HPC center when a single user runs a parameter sweep on a scientific or engineering simulation. Each of the submitted jobs is substantially similar, differing only in the input parameters. Furthermore, we simulate two extreme cases: (a) when all the jobs are difficult to co-schedule (8 MB working set each); and (b) when all jobs are easy to co-schedule (0.5 MB working set each).

We configure the synthetic benchmark for a fixed working set size and restrict it to execute using a single thread, and run 140 instances of the benchmark under AIFD and Slurm. All instances are submitted at the beginning of execution.

The working set size for each job was fixed at 0.5 MB. Since the working set of the applications is small (compared to the L3 cache size), AIFD dynamically over-commits the system and allows up to 52 threads to run simultaneously.

This is in contrast to the case of a working set size of 8 MB. In this case, even at a zero over-commit level, the co-scheduled applications experience a degradation of up to 80%.

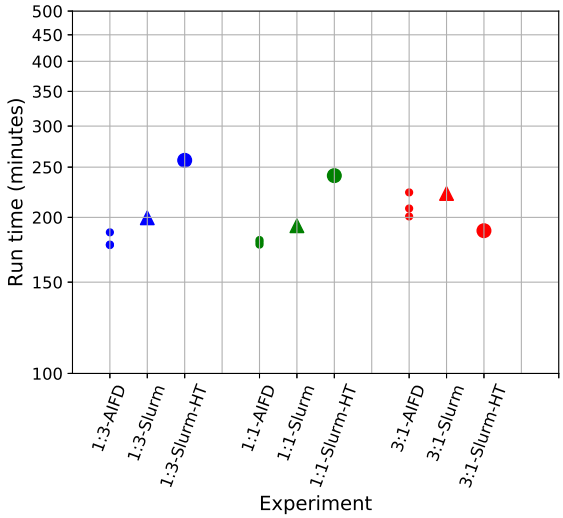


Figure 6.5: Heterogeneous workload with a mixture of synthetic benchmarks of two different working set sizes — 0.5 MB and 8 MB. Results are shown for three different proportions of the two job types: 1:3, 1:1, and 3:1 (Running time: AIFD, Slurm, and Slurm-HT for different workloads with synthetic benchmarks).

AIFD detects this degradation in throughput and under-commits the system.

Figure 6.4 shows end-to-end running time for the two cases described above. Slurm runs consist of two sets of experiments: one with hyper-threading turned off (Slurm-1.0), and the other where Slurm is configured to use all the hyper-threaded cores, i.e., twice the number of physical cores (Slurm-HT). We observe that in the 0.5 MB-homogeneous case, AIFD performs 5% worse than Slurm-HT and 9% better than Slurm-1.0. As expected, Slurm-1.0 achieves the worst throughput in this extreme case since it limits the number of simultaneously executing jobs to the number of physical cores. In the 8 MB-homogeneous case, AIFD outperforms Slurm-1.0 by 5% and outperforms Slurm-HT by 23%.

6.4.3 Synthetic benchmarks: Heterogeneous case

Next, we simulate multiple users with distinct applications. We evaluate the AIFD algorithm by varying the working set sizes and the frequency of accessing the working set of the synthetic benchmark. We also vary the percentage of different jobs in the system.

For the first experiment in this category, we use two different types of synthetic benchmarks in the same run: one with a 0.5 MB working set, and the other with an 8 MB

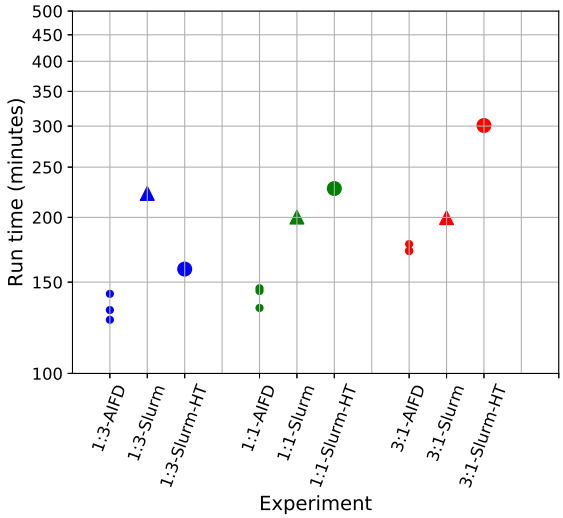


Figure 6.6: Heterogeneous workload with a mixture of synthetic benchmarks of two different types, memory-intensive and CPU-intensive, for 8 MB working set size. Results are shown for three different proportions of the two job types: 1:3, 1:1, 3:1 (Running time: AIFD, Slurm, and Slurm-HT for different workloads with synthetic benchmarks).

working set, and vary the proportion of the jobs from the two types. Figure 6.5 shows end-to-end running time for three different proportions (1:3, 1:1, and 3:1) under AIFD and Slurm.

We observe that AIFD is better than Slurm by 20% on average. In the 3:1 case, where the number of 0.5 MB jobs was 75% of the workload, AIFD is 10% worse than the best configuration of Slurm (Slurm-HT) and is 5% better on average than Slurm-1.0. On average, AIFD is 18% better than Slurm. To understand the reasons for this improvement, we analyze the actual throughput, the number of running threads, and the number of jobs with large working set in the system for the 1:1 case during the run. Figure 6.7 shows the four different parameters: actual throughput, ideal throughput, number of running threads, and the number of threads with large working set, over time for the 1:1 case. While AIFD recognizes the degradation when the number of contentious jobs in the system increase, Slurm, with its fixed configuration, is unable to adjust the level of over-commitment and hence, suffers from severe degradation.

The second experiment in this category introduces jobs with varying intensity of access to the memory. This simulates the case of CPU-intensive applications, where many

CPU-intensive operations follow each access to memory operation. Therefore, we modify the synthetic benchmark to execute a fixed the number of CPU operations per memory access.

We observe that AIFD outperforms Slurm in all the cases (73% in the best case, and 15% in the worst case). We analyze the most interesting case of jobs with 8 MB working set size where the proportion of memory- to CPU-intensive jobs was 1:3. Although the total working set size of even 9 co-scheduled jobs ($9 \times 8\ MB = 72\ MB$) in this case exceeds the L3 cache size (70 MB in total), the rate of access to the working set dictates the behavior of the AIFD algorithm. This results in a low degradation in actual throughput (compared to the ideal throughput), and therefore, allows AIFD to over-commit the CPU cores and co-schedule up to 56 jobs. Note that while Slurm-HT achieves a better throughput than Slurm-1.0 in this case, it is still worse than AIFD. The reason is that the presence of memory-intensive jobs, although in a smaller number, requires the system to dynamically lower the over-commit level. Slurm-HT has a fixed configuration and cannot adjust the over-commit level dynamically.

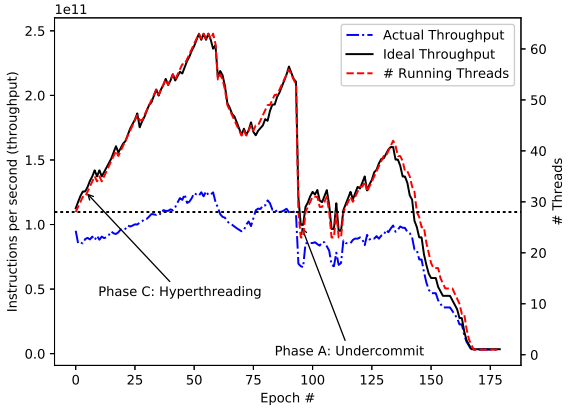


Figure 6.7: Different phases of AIFD for the heterogeneous workload as shown by the 1:1-AIFD case in Figure 6.5.

6.4.4 Synthetic benchmarks: Multiple threads

For the last set of evaluations with the synthetic benchmark, we further modify the benchmark to use multiple threads for accessing data from its working set or executing instructions on the CPU. Table 6.2 summarizes the different sets of experiments in this category. Note that while the benchmark used is synthetic, the pre-conditions become

Exper.	Working Set Size	No. of Threads	CPU- or Memory-intensive	Job Length	Arrival Time
E1	Random	Random	CPU	Same	Same
E2	Random	Random	Memory	Same	Same
E3	Random	Random	Random	Same	Same
E4	Random	Random	Random	Random	Same
E5	Random	Random	Random	Random	Expon. Dist.

Table 6.2: Summary of experiments and pre-conditions in Section 6.4.4. The working set size for a job is chosen at random from the following set: $\{0.5\ MB, 8\ MB\}$; number of threads for a job is chosen at random from the following set: $\{1, 2, 4, 6, 8\}$; and the job lengths were assigned randomly from the following set: $\{5\ mins., 10\ mins., 15\ mins., \ldots, 40\ mins.\}$. 140 jobs were used for experiments E1-E3; for experiments E4 and E5, 241 jobs were used.

increasingly typical of the real-world scenarios when progressing from E1 through E5. Exponential distribution is used to assign the inter-arrival times for the jobs in the workload for experiment E5.

Figure 6.8 shows end-to-end running time for the different experiments (E1 through E5) with AIFD and with Slurm. We note that AIFD is equal to Slurm in the worst case, and by 164% in the best case.

Next, we analyze the results for experiments E4. Figure 6.10 shows three different parameters (throughput, runtime, and number of running threads) over running time. We observe that the system goes through three different regimes: under-commit, over-commit, and hyper-threading, at different points in time.

In the under-commit phase (starting at the 290^{th} minute), first, the system settles on a plateau with 16 threads; then the system, once a CPU-intensive job is replaced by memory-intensive job (8 MB WSS), starts under-committing further until it settles on the second plateau with 8 threads.

We note that as a large number of MemLarge start completing (towards the end of the 304^{th} minute) and are replaced by CPU-intensive jobs, AIFD starts over-committing the CPU cores until it reaches a plateau at 52 threads (in the 320^{th} minute). The plateau is governed by the input parameters to the algorithm: the base degradation threshold and the over-commit factors. The parameters disallow further over-commit, which would have lead to excessive performance degradation.

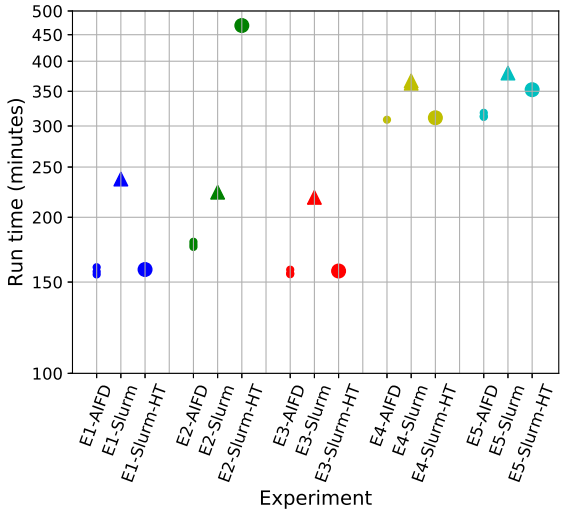


Figure 6.8: Running time: AIFD, Slurm, and Slurm-HT for the different configurations shown in Table 6.2.

To understand the impact of the aging policy (described in Section 6.4.1) and the effect of AIFD on the execution times of the applications, we analyze the turnaround times, waiting time, and the execution times for all the jobs in experiment E5.

Turnaround time for a job is defined as the time between when the job was first submitted and when it completed. Waiting time is the time spent by a job in the queue waiting for execution. In the case of AIFD, the waiting time includes the time spent in the batch pool when a job is checkpointed and removed from execution by the algorithm. Note that the waiting time for a job can be potentially longer under AIFD than Slurm. The execution time of a job is defined as the time it spends in actual execution.

Figure 6.11 shows these results for the best configuration of Slurm for this workload (Slurm-HT) and AIFD. As expected, the variation in the average waiting times for all the jobs under AIFD is higher than Slurm. However, the median turnaround times and the execution times for all the jobs under AIFD are 5% better than Slurm. This is in part due the higher throughput (and lower contention for shared resources) under AIFD. Aging policy helps keep the waiting times under AIFD low.

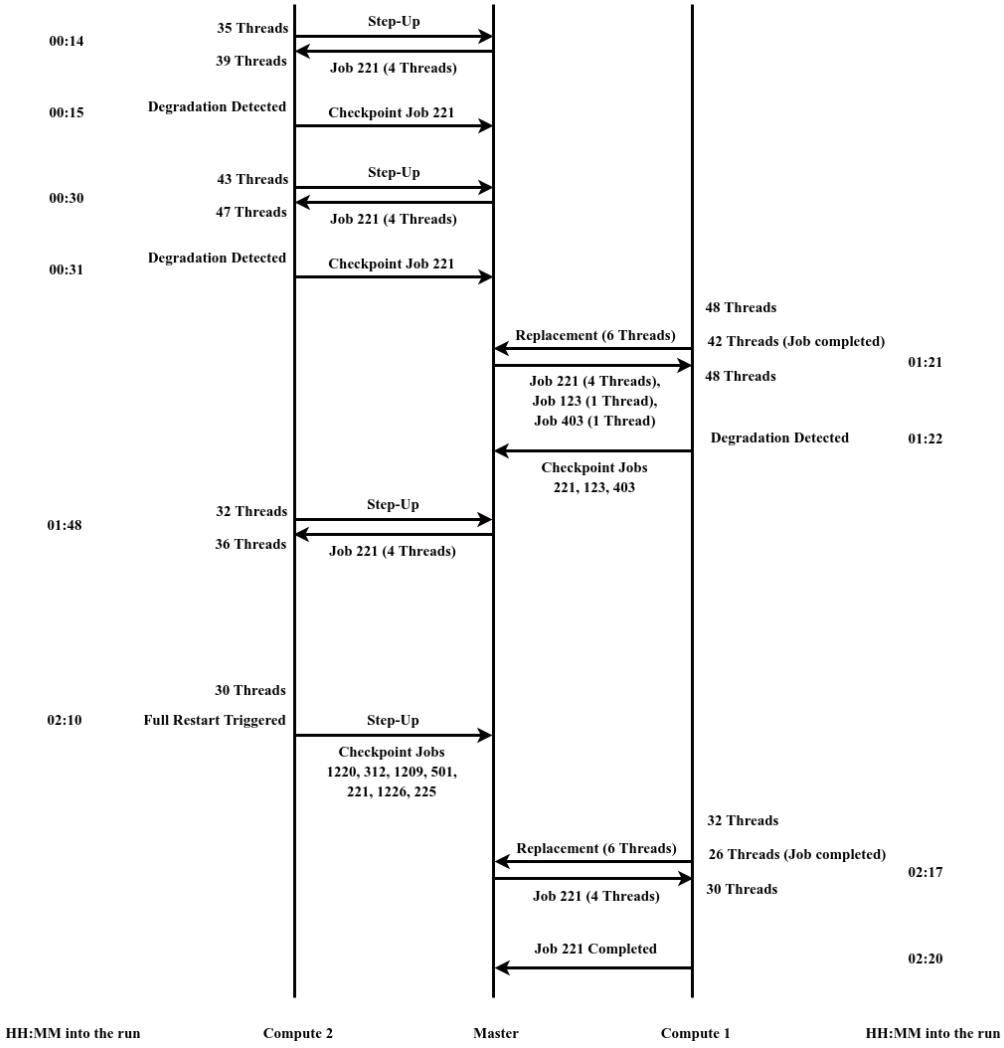


Figure 6.9: Timeline presenting node re-balancing in a multi-node setup.

6.4.5 Multi-node Re-balancing

For this experiment, we deployed a three node setup with a shared filesystem: a master controlling node, and two compute nodes. Incoming jobs are submitted to the master, and the master later delegates the jobs to either of the two compute nodes. The compute nodes run independent AIFD instances that interact with the master to schedule jobs.

As will be seen, jobs gradually and automatically migrate between the two compute nodes according to whether they are CPU-intensive or RAM-intensive. Note that the AIFD instances on the two compute nodes are not synchronized and may be executing different AIFD phases at the same time. The master ensures that a job is not scheduled simultaneously

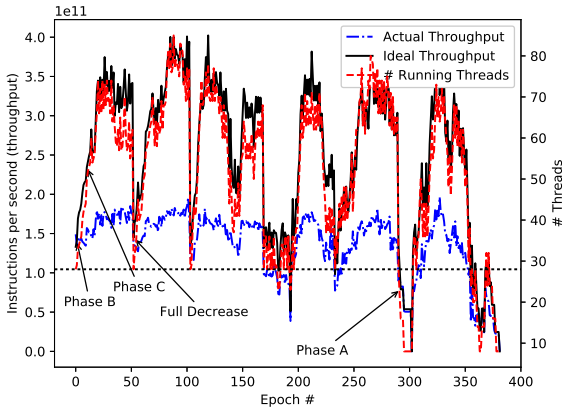


Figure 6.10: Different phases of AIFD for the heterogeneous workload as shown by the E4 case in Table 6.2.

on both the nodes. For a job to migrate from one node to the other, it must checkpoint to the shared filesystem from one node and be restarted on the other node.

Figure 6.9 presents a timeline demonstrating the life cycle of a 4-threaded CPU-intensive job, job id 221 (Resident Set Size: 872 KB and Virtual Memory Size: 232 MB). Job id 221 migrates between the compute nodes, as the AIFD instance on each compute nodes tries to re-balance its respective running job sets. As shown in Figure 6.9, job id 221 is initially scheduled on Compute-2 (fourteen minutes into the run), when the AIFD instance on Compute-2 attempts to over-commit in the HT-enabled phase (with 35 total job threads already running). AIFD instance on Compute-2 observes a degradation in the system throughput after scheduling job id 221 and decides to de-schedule the job. Thirty minutes into the run, job s 221 is re-scheduled on Compute 2 but was re-pooled for the same reason.

One hour and twenty one minutes into the run, a 6-threaded job completes on Compute-1. Three jobs, including job 221, are scheduled on Compute-1 to fill in the available slots. The AIFD instance on Compute-1 observes a degradation after this replacement and decides to re-pool the three new jobs.

Later, job id 221 is scheduled again on Compute-2 (1 hour 48 minutes into the run). Adding job id 221 to the current job set results in an improved throughput, and AIFD continues with the new job set for the following epochs, until a full restart is triggered at 2 hours and 10 minutes. Finally, job id 221 migrates back to Compute-1 at 2 hours and

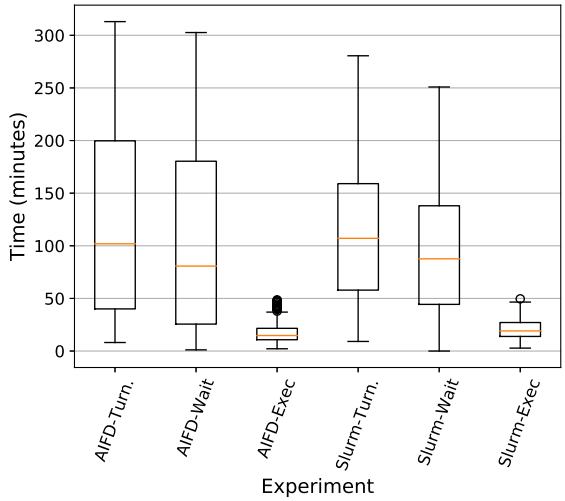


Figure 6.11: Boxplot for turnaround time, waiting time, and execution time for the E5 workload under AIFD and Slurm-HT.

17 minutes and completes at 2 hours and 20 minutes.

We observe that as the two independent AIFD instances strive to achieve an optimal throughput on the two individual nodes, the jobs are *automatically migrated* into two sets of compatible jobs. A set of mutually compatible jobs is not fixed and changes over time, as jobs migrate between the two nodes. Thus, CPU-intensive jobs gradually migrate to Compute-1, while RAM-intensive jobs gradually migrate to Compute-2.

6.4.6 Real-world benchmarks

In this section, we compare the AIFD algorithm against Slurm, running twelve multi-threaded applications (see Table 6.3) from four HPC benchmark suites. The experiments were run with 200 jobs where each job was picked at random from the twelve applications, ensuring equal representation for all the twelve applications. Each job was assigned the number of threads at random from the set: $\{1, 2, 4, 6, 8, 16\}$. Each job was configured to run for forty minutes to an hour (by changing the number of iterations of the main compute loop in the application). In contrast to the synthetic benchmark, the working set sizes of the real-world applications are not known in advance. Unlike synthetic benchmarks, real-world applications had large working set sizes (upto 750 MB).

Figure 6.12 shows the end-to-end running time of the real-world workload under

Application	Benchmark suite	Remark
IS	NAS OpenMP	Version 3.3.1
EP	NAS OpenMP	Version 3.3.1
CG	NAS OpenMP	Version 3.3.1
Raytrace	SPLASH2	
Volrend	SPLASH2	
LU	SPLASH2	Contiguous and Non-contiguous
Water-Nsquared	SPLASH2	
Blackscholes	PARSEC	Version 3.0
Ferret	PARSEC	Version 3.0
Swaptions	PARSEC	Version 3.0
Probe	Stencil Probe	

Table 6.3: HPC applications used for evaluation.

AIFD and different Slurm configurations. AIFD performs better than Slurm-1.0 by 75% and Slurm-HT by 12%.

We observe that the workload with real-world applications is similar to the E1 case in Section 6.4.4 in that the applications are CPU-intensive and don't degrade the performance of other co-scheduled applications. This allows AIFD to over-commit the CPU cores by running up to 72 threads under hyper-threading regime for several epochs.

6.4.7 Sensitivity analysis

Finally, we evaluate the sensitivity of the proposed algorithm on the different predefined input parameters and architecture. We run experiment E4 (Section 6.4.3, Table 6.2) under AIFD for different base degradation thresholds. The variation in the running times for thresholds between 0.4 to 0.8 is less than 10%.

In the interest of space, we omit the results for the other input parameters here. We note, however, that the results are similar and the variation is less than 5% in all cases.

Remark: As we vary the predefined parameters, the same type of results hold. This shows that there is a broad plateau, and the results are not engineered toward one architecture or another. To further validate the stability of our results, we conducted two experiments with synthetic benchmarks on two different architectures (Sys-2 and Sys-3 in Section 6.4.1, Table 6.1). We observed that AIFD is equivalent to the best configuration of Slurm on Sys-2 and is 10% better than the best configuration of Slurm on Sys-3.

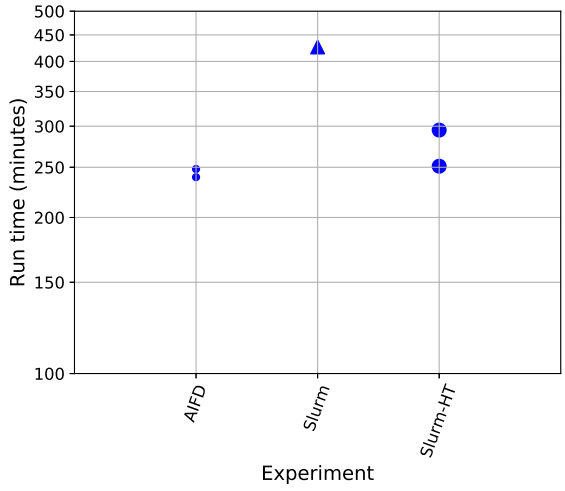


Figure 6.12: Running time: AIFD, Slurm, and Slurm-HT for a workload with real-world HPC applications.

Chapter 7

Conclusion and Future Directions

Bibliography

[1] D. Anderson, M. Hibler, L. Stoller, T. Stack, and J. Lepreau, "Automatic online validation of network configuration in the Emulab network testbed," in *IEEE International Conference on Autonomic Computing*, USA, 2006, pp. 134–142.

[2] Openstack, "Ironic." [Online]. Available: http://docs.openstack.org/developer/ironic/deploy/user-guide.html

[3] OpenCrowbar, "The Crowbar project," 2015. [Online]. Available: https://opencrowbar.github.io

[4] Canonical, "Metal as a service (MAAS)," 2015. [Online]. Available: http://maas.ubuntu.com/docs/

[5] Puppetlabs, "Provisioning with Razor." [Online]. Available: https://docs.puppetlabs.com/pe/latest/razor_intro.html

[6] Cobbler, "Cobbler." [Online]. Available: https://cobbler.github.io

[7] A. Chandrasekar and G. Gibson, "A comparative study of baremetal provisioning frameworks," Parallel Data Laboratory, Carnegie Mellon University, Tech. Rep. CMU-PDL-14-109, 2014.

[8] Foreman, "Foreman provisioning and configuration system." [Online]. Available: http://theforeman.org

[9] D. Van der Veen *et al.*, "Openstack Ironic Wiki." [Online]. Available: https://wiki.openstack.org/wiki/Ironic

[10] Y. Omote, T. Shinagawa, and K. Kato, "Improving agility and elasticity in bare-metal clouds," in *Proceedings of the Twentieth International Conference on Architectural*

Support for Programming Languages and Operating Systemsi, ASPLOS. Turkey: ACM, 2015, pp. 145–159.

[11] Y. Klimenko, "Technique for reliable network booting of an operating system to a client computer," Oct. 26 1999, uS Patent 5,974,547. [Online]. Available: https://www.google.com/patents/US5974547

[12] D. Sposato, "Method and apparatus for remotely booting a client computer from a network by emulating remote boot chips," Oct. 8 2002, uS Patent 6,463,530. [Online]. Available: https://www.google.com/patents/US6463530

[13] C. Haun, C. Prouse, J. Sokol, and P. Resch, "Providing a reliable operating system for clients of a net-booted environment," Jun. 15 2004, uS Patent 6,751,658. [Online]. Available: https://www.google.com/patents/US6751658

[14] K. Salah, R. Al-Shaikh, and M. Sindi, "Towards green computing using diskless high performance clusters," in *Network and Service Management (CNSM), 2011 7th International Conference on*. IEEE, 2011, pp. 1–4.

[15] B. Guler, M. Hussain, T. Leng, and V. Mashayekhi, "The advantages of diskless hpc clusters using nas," *Technical Report Dell Power Solutions*, 2002.

[16] D. Daly, J. H. Choi, J. E. Moreira, and A. Waterland, "Base operating system provisioning and bringup for a commercial supercomputer," in *Parallel and Distributed Processing Symposium, (IPDPS)*. IEEE, 2007, pp. 1–7.

[17] R. Minnich, J. Hendricks, and D. Webster, "The linux bios," in *Proceedings of the 4th Annual Linux Showcase and Conference*. USENIX, 2000.

[18] R. Lewis, "Virtual disk image system with local cache disk for iscsi communications," Aug. 2 2005, uS Patent 6,925,533. [Online]. Available: https://www.google.com/patents/US6925533

[19] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, USA, 2008, pp. 161–174.

[20] M. Nelson, B.-H. Lim, G. Hutchins *et al.*, "Fast transparent migration for virtual machines." in *USENIX Annual technical conference, general track*, USA, 2005, pp. 391–394.

[21] "The clever 'DOUBLEAGENT' attack turns antivirus into malware," https://www.wired.com/2017/03/clever-doubleagent-attack-turns-antivirus-malware/.

[22] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Vmm-based hidden process detection and identification using lycosid," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2008, pp. 91–100.

[23] "Amazon. summary of the october 22,2012 aws service event in the us-east region." https://aws.amazon.com/message/680342.

[24] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 conference on internet measurement conference*. ACM, 2014, pp. 475–488.

[25] T. Fox-Brewster, "What is the shellshock bug? is it worse than heartbleed," *The Guardian*, 2014.

[26] R. H. Inc., "GHOST: glibc vulnerability (CVE-2015-0235)," https://access.redhat.com/articles/1332213.

[27] D. A. Wheeler, "Flawfinder," 2011.

[28] C. C. Security, "Rough Auditing Tool for Security," https://security.web.cern.ch/security/recommendations/en/codetools/rats.shtml.

[29] J. Viega, J.-T. Bloch, Y. Kohno, and G. McGraw, "Its4: A static vulnerability scanner for c and c++ code," in *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*. IEEE, 2000, pp. 257–267.

[30] J. S. Foster and A. S. Aiken, "Type qualifiers: lightweight specifications to improve software quality," Ph.D. dissertation, Citeseer, 2002.

[31] W. Jimenez, A. Mammar, and A. Cavalli, "Software vulnerabilities, prevention and detection methods: A review1," *Security in Model-Driven Architecture*, p. 6, 2009.

[32] P. Szor, *The art of computer virus research and defense.* Pearson Education, 2005.

[33] M. Davis, S. Bodmer, and A. LeMasters, *Hacking Exposed Malware and Rootkits.* McGraw-Hill, Inc., 2009.

[34] A. Emigh, "The crimeware landscape: Malware, phishing, identity theft and beyond," *Journal of Digital Forensic Practice*, vol. 1, no. 3, pp. 245–260, 2006.

[35] "checkrootkit," http://www.chkrootkit.org/.

[36] "Linux Malware Detect," https://www.rfxn.com/projects/linux-malware-detect/.

[37] "Clam AntiVirus," https://www.clamav.net.

[38] N. Idika and A. P. Mathur, "A survey of malware detection techniques," *Purdue University*, vol. 48, 2007.

[39] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16).* Savannah, GA: USENIX Association, 2016, pp. 249–264. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gao

[40] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 350–361.

[41] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Towards a next generation data center architecture: scalability and commoditization," in *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow.* ACM, 2008, pp. 57–62.

[42] S. Legtchenko, H. Williams, K. Razavi, A. Donnelly, R. Black, A. Douglas, N. Cheriere, D. Fryer, K. Mast, A. D. Brown, A. Klimovic, A. Slowey, and A. Rowstron, "Understanding rack-scale disaggregated storage," in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17).* Santa Clara, CA: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/hotstorage17/program/presentation/legtchenko

[43] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, "Flash storage disaggregation," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 29.

[44] E. K. Lee and C. A. Thekkath, "Petal: Distributed virtual disks," in *ACM SIGPLAN Notices*, vol. 31, no. 9. ACM, 1996, pp. 84–92.

[45] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand, "Parallax: Managing storage for a million machines." in *HotOS*, 2005.

[46] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "Legoos: A disseminated, distributed OS for hardware resource disaggregation," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 69–87. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/shan

[47] A. Mohan, A. Turk, R. S. Gudimetla, S. Tikale, J. Hennesey, U. Kaynar, G. Cooperman, P. Desnoyers, and O. Krieger, "M2: Malleable metal as a service," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, April 2018, pp. 61–71.

[48] A. Mosayyebzadeh, A. Mohan, S. Tikale, M. Abdi, N. Schear, T. Hudson, C. Munson, L. Rudolph, G. Cooperman, P. Desnoyers, and O. Krieger, "Supporting security sensitive tenants in a bare-metal cloud," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 587–602. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/mosayyebzadeh

[49] "OpenStack Bare Metal Provisioning Program," https://wiki.openstack.org/wiki/Ironic.

[50] D. Clerc, L. Garcés-Erice, and S. Rooney, "Os streaming deployment," in *Performance Computing and Communications Conference (IPCCC), 2010 IEEE 29th International*. IEEE, 2010, pp. 169–179.

[51] Y. Shin and L. Williams, "Is complexity really the enemy of software security?" in *Proceedings of the 4th ACM workshop on Quality of protection*. ACM, 2008, pp. 47–50.

[52] T. Mccabe, "More complex= less secure," *McCabe Software, Inc*, p. 12, 2014.

[53] D. Williams, R. Koller, and B. Lum, "Say goodbye to virtualization for a safer cloud," in *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. Boston, MA: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/hotcloud18/presentation/williams

[54] F. Oliveira, S. Suneja, S. Nadgowda, P. Nagpurkar, and C. Isci, "Opvis: extensible, cross-platform operational visibility and analytics for cloud," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track*. ACM, 2017, pp. 43–49.

[55] R. Koller, C. Isci, S. Suneja, and E. De Lara, "Unified monitoring and analytics in the cloud," in *7th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[56] H. wook Baek, A. Srivastava, and J. Van der Merwe, "Cloudvmi: Virtual machine introspection as a cloud service," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE, 2014, pp. 153–158.

[57] F. Yao and R. H. Campbell, "Cryptvmi: Encrypted virtual machine introspection in the cloud," in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*. IEEE, 2014, pp. 977–978.

[58] L. Jia, M. Zhu, and B. Tu, "T-vmi: Trusted virtual machine introspection in cloud environments," in *Cluster, Cloud and Grid Computing (CCGRID), 2017 17th IEEE/ACM International Symposium on*. IEEE, 2017, pp. 478–487.

[59] Amazon, "Amazon Inspector," https://aws.amazon.com/inspector/.

[60] S. Nadgowda, C. Isci, and M. Bal, "DÉjÀvu: Bringing black-box security analytics to cloud," in *Proceedings of the 19th International Middleware Conference Industry*, ser. Middleware '18. New York, NY, USA: ACM, 2018, pp. 17–24. [Online]. Available: http://doi.acm.org/10.1145/3284028.3284031

[61] S. Nadgowda and C. Isci, "Drishti: Disaggregated and interoperable security analytics framework for cloud," in *Proceedings of the ACM Symposium on Cloud Computing*,

ser. SoCC '18. New York, NY, USA: ACM, 2018, pp. 528–528. [Online]. Available: http://doi.acm.org/10.1145/3267809.3275470

[62] S. Suneja, R. Koller, C. Isci, E. de Lara, A. Hashemi, A. Bhattacharyya, and C. Amza, "Safe inspection of live virtual machines," in *ACM SIGPLAN Notices*, vol. 52, no. 7. ACM, 2017, pp. 97–111.

[63] IBM, "IBM bigFix: A collaborative endpoint management and security platform," .

[64] S. Corp., "Symantec Endpoint Protection," https://www.symantec.com/smb/endpoint-protection.

[65] Tanium, "Platform for endpoint management and security."

[66] "Open source tools for container security and compliance." https://anchore.com.

[67] "Automatic container vulnerability and security scanning for appc and docker," https://coreos.com/clair/docs/latest/.

[68] "Container security - docker, kubernetes, openshift, mesos." https://www.aquasec.com/.

[69] "Docker security and container security platform." https://twistlock.com.

[70] S. Nadgowda, S. Duri, C. Isci, and V. Mann, "Columbus: Filesystem tree introspection for software discovery," in *Cloud Engineering (IC2E), 2017 IEEE International Conference on.* IEEE, 2017, pp. 67–74.

[71] M. Banikazemi, D. Poff, and B. Abali, "Storage-based intrusion detection for storage area networks (sans)," in *Mass Storage Systems and Technologies, 2005. Proceedings. 22nd IEEE/13th NASA Goddard Conference on.* IEEE, 2005, pp. 118–127.

[72] M. Allalouf, M. Ben-Yehuda, J. Satran, and I. Segall, "Block storage listener for detecting file-level intrusions," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on.* IEEE, 2010, pp. 1–12.

[73] I. Amazon Web Services, "Vmware cloud in aws," 2020. [Online]. Available: https://aws.amazon.com/vmware/

[74] D. Clerc, L. Garcés-Erice, and S. Rooney, "Os streaming deployment," in *International Performance Computing and Communications Conference*, Dec 2010, pp. 169–179.

[75] A. Mosayyebzadeh, G. Ravago, A. Mohan, A. Raza, S. Tikale, N. Schear, T. Hudson, J. Hennessey, N. Ansari, K. Hogan, C. Munson, L. Rudolph, G. Cooperman, P. Desnoyers, and O. Krieger, "A secure cloud with minimal provider trust," in *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. Boston, MA: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/hotcloud18/presentation/mosayyebzadeh

[76] M. Bhadauria and S. A. McKee, "An approach to resource-aware co-scheduling for CMPs," in *Proc. of the 24th ACM Int. Conf. on Supercomputing*. ACM, 2010, pp. 189–199.

[77] E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez, "Adaptive parallel job scheduling with flexible CoScheduling," *Parallel and Distributed Systems, IEEE Trans. on*, vol. 16, no. 11, pp. 1066–1077, 2005.

[78] Y. Jiang and X. Shen, "Exploration of the influence of program inputs on CMP co-scheduling," in *Euro-Par 2008 — Parallel Processing*. Springer, 2008, pp. 263–273.

[79] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, "Measuring interference between live datacenter applications," in *Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society Press, 2012, p. 51.

[80] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. of the 42nd Annual Int. Symp. on Computer Architecture*. ACM, 2015, pp. 450–462.

[81] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proc. of the 44th annual IEEE/ACM Int. Symp. on Microarchitecture*. ACM, 2011, pp. 248–259.

[82] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "Contention aware execution: Online contention detection and response," in *Proc. of the 8th annual IEEE/ACM Int. Symp. on Code Generation and Optimization*. ACM, 2010, pp. 257–265.

[83] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *Computer Architecture (ISCA), 2011 38th Annual Int. Symp. on*. IEEE, 2011, pp. 283–294.

[84] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa, "Reqos: Reactive static/dynamic compilation for QoS in warehouse scale computers," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 89–100.

[85] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune, "Optimizing Google's warehouse scale computers: The NUMA experience," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th Int. Symp. on*. IEEE, 2013, pp. 188–197.

[86] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 607–618.

[87] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "CPI^2: CPU performance isolation for shared compute clusters," in *Proc. of the 8th ACM European Conf. on Computer Systems (EuroSys'13)*. ACM, 2013, pp. 379–391.

[88] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "Smite: Precise QOS prediction on real-system SMT processors to improve utilization in warehouse scale computers," in *Proc. of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 406–418.

[89] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 77–88, 2013.

[90] ——, "Quasar: Resource-efficient and QoS-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[91] S. Niu, J. Zhai, X. Ma, M. Liu, Y. Zhai, W. Chen, and W. Zheng, "Employing checkpoint to improve job scheduling in large-scale systems," in *Job Scheduling Strategies for Parallel Processing (JSSPP)*. Springer, 2013, pp. 36–55.

[92] F. Liu and J. B. Weissman, "Elastic job bundling: An adaptive resource request strategy for large-scale parallel applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. ACM, 2015.

[93] D. Klusáček, H. Rudová, and M. Jaroš, "Multi resource fairness: Problems and challenges," in *Job Scheduling Strategies for Parallel Processing (JSSP)*. Springer, 2014, pp. 81–95.

[94] F. Negele, F. Friedrich, and B. Egger, "On the design and implementation of an efficient lock-free scheduler," 2015.

[95] S. Prabhakaran, M. Neumann, S. Rinke, F. Wolf, A. Gupta, and L. V. Kale, "A batch system with efficient adaptive scheduling for malleable and evolving applications," in *Parallel and Distributed Processing Symposium (IPDPS'15)*. IEEE, 2015, pp. 429–438.

[96] D. Seo, M. Kim, H. Eom, and H. Y. Yeom, "Bubble task: A dynamic execution throttling method for multi-core resource management," in *Job Scheduling Strategies for Parallel Processing (JSSP)*. Springer, 2014, pp. 1–16.

[97] O. Shai, E. Shmueli, and D. G. Feitelson, "Heuristics for resource matching in Intel's compute farm," in *Job Scheduling Strategies for Parallel Processing (JSSP)*. Springer, 2014, pp. 116–135.

[98] D. Talby and D. G. Feitelson, "Improving and stabilizing parallel computer performance using adaptive backfilling," in *Parallel and Distributed Processing Symposium, 2005. Proc. 19th IEEE Int.* IEEE, 2005, pp. 84a–84a.

[99] ZDNet, "Facebook: Virtualisation does not scale," 2011. [Online]. Available: http://www.zdnet.com/article/facebook-virtualisation-does-not-scale/

[100] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 6, pp. 931–945, 2011.

[101] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. USA: ACM, 2009, pp. 199–212.

[102] Softlayer, "SoftLayer to outflank rivals with bare metal, InfiniBand, and Power8," 2014. [Online]. Available: https://www.enterprisetech.com/2014/07/30/softlayer-outflank-rivals-bare-metal-infiniband-power8/

[103] S. K. Barker and P. Shenoy, "Empirical evaluation of latency-sensitive application performance in the cloud," in *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*. USA: ACM, 2010, pp. 35–46.

[104] S. M. Trimberger and J. J. Moore, "FPGA security: Motivations, features, and applications," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1248–1265, 2014.

[105] C. Maurice, C. Neumann, O. Heen, and A. Francillon, *Confidentiality Issues on a GPU in a Virtualized Environment*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 119–135.

[106] AWS, 2017. [Online]. Available: https://aws.amazon.com/hpc/

[107] ——, "Amazon EC2 elastic GPUs," 2017. [Online]. Available: https://aws.amazon.com/hpc/

[108] Cirrascale, "Cloud services for deep learning," 2017. [Online]. Available: http://www.cirrascale.com/cloud/

[109] D. Schatzberg, J. Cadden, H. Dong, O. Krieger, and J. Appavoo, "Ebbrt: A framework for building per-application library operating systems." in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 671–688.

[110] Softlayer, "Big data solutions," 2015. [Online]. Available: http://www.softlayer.com/big-data

[111] Rackspace, "Rackspace cloud big data OnMetal," 2015. [Online]. Available: http://go.rackspace.com/baremetalbigdata/

[112] ChameleonCloud, "Chameleon cloud: Bare metal user guide," 2015. [Online]. Available: https://www.chameleoncloud.org/docs/bare-metal-user-guide-old/

[113] Internap, "Bare-Metal AgileSERVER," 2015. [Online]. Available: http://www.internap.com/bare-metal/

[114] (2018) OpenQRM: http://www.openqrm.org/. [Online]. Available: http://www.openqrm.org/index-1.html

[115] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. USA: USENIX, 2006, pp. 307–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298485

[116] ——, "Ceph storage," 2017. [Online]. Available: http://ceph.com/ceph-storage/

[117] J. Hennessey, S. Tikale, A. Turk, E. U. Kaynar, C. Hill, P. Desnoyers, and O. Krieger, "HIL: Designing an exokernel for the data center," in *Proceedings of the Seventh ACM Symposium on Cloud Computing, (SOCC)*. USA: ACM, 2016, pp. 155–168. [Online]. Available: https://doi.acm.org/10.1145/2987550.2987588

[118] M. Chadalapaka, J. Satran, K. Meth, and D. Black, "Internet Small Computer System Interface (iSCSI) Protocol (Consolidated)," RFC 7143 (Proposed Standard), RFC Editor, USA, pp. 1–295, 2014. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7143.txt

[119] WMware.com, "VMware Workstation 5.0 Understanding Clones." [Online]. Available: https://www.vmware.com/support/ws5/doc/ws_clone_overview.html

[120] S. Hogg, "Clos networks: What's old is new again," Jan 2014. [Online]. Available: https://www.networkworld.com/article/2226122/cisco-subnet/clos-networks--what-s-old-is-new-again.html

[121] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, "Flash storage disaggregation," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 29.

[122] S. A. Weil, *Ceph: reliable, scalable, and high-performance distributed storage*, 2007, vol. 68, no. 11.

[123] "About the lustre file system." [Online]. Available: http://lustre.org/about/

[124] J. Bonwick and B. Moore, "ZFS: The last word in file systems," 2007. [Online]. Available: https://wiki.illumos.org/download/attachments/1146951/zfs_last.pdf

[125] (2018) Btrfs: https://btrfs.wiki.kernel.org. [Online]. Available: https://btrfs.wiki.kernel.org/index.php/Main_Page

[126] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The linux b-tree filesystem," *Trans. Storage*, vol. 9, no. 3, pp. 9:1–9:32, Aug. 2013. [Online]. Available: http://doi.acm.org/10.1145/2501620.2501623

[127] D. Teigland and H. Mauelshagen, "Volume managers in Linux." in *USENIX Annual Technical Conference, FREENIX Track*, 2001, pp. 185–197. [Online]. Available: http://static.usenix.org/legacy/events/usenix01/freenix01/full_papers/teigland/teigland_html/

[128] OpenStack, "Ceph IO Performance." [Online]. Available: https://docs.openstack.org/performance-docs/latest/test_results/ceph_testing/index.html#ceph-rbd-performance-results-50-osd

[129] V. Inc, "iSCSI Performance Depends on Storage Performance." [Online]. Available: https://docs.vmware.com/en/VMware-vSphere/6.0/com.vmware.vsphere.storage.doc/GUID-548C8064-23DB-44EB-8FFC-BFEF5D39DA3A.html

[130] F. Tomonori and M. Christie, "tgt: Framework for storage target drivers," in *Linux Symposium*, 2006.

[131] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, 2014. [Online]. Available: https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment

[132] mcb30, "iPXE: Open Source Boot Firmware," 2015. [Online]. Available: http://ipxe.org

[133] Microsoft, "About iSCSI Boot," 2009. [Online]. Available: https://technet.microsoft.com/en-us/library/ee619722(v=ws.10).aspx

[134] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "A large-scale study of flash memory failures in the field," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 1. ACM, 2015, pp. 177–190.

[135] ——, "Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field," in *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*. IEEE, 2015, pp. 415–426.

[136] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[137] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010.

[138] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The nas parallel benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[139] "Rally Benchmarking Tool for OpenStack." [Online]. Available: https://wiki.openstack.org/wiki/Rally

[140] A. Kopytov, "Sysbench: a system performance benchmark," *URL: http://sysbench.sourceforge. net*, 2004.

[141] ——, "Sysbench manual," *MySQL AB*, 2012.

[142] A. E. C. Cloud, "Amazon web services," *Retrieved November*, vol. 9, p. 2011, 2011.

[143] D. Aguado, T. Andersen, A. Avetisyan, J. Budnik, M. Criveti, A. Doroiman, A. Hoppe, G. Menegaz, A. Morales, A. Moti *et al.*, *A practical approach to cloud IaaS with IBM SoftLayer: Presentations guide*. IBM Redbooks, 2016.

[144] "Common Vulnerability Exposures," https://cve.mitre.org/.

[145] "National Vulnerability Database," https://nvd.nist.gov/.

[146] W. Yan and N. Ansari, "Why anti-virus products slow down your machine?" in *Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on.* IEEE, 2009, pp. 1–6.

[147] "Creating a Classified Processing Enclave in the Public Cloud |IARPA," https://www.iarpa.gov/index.php/working-with-iarpa/requests-for-information/creating-a-classified-processing-enclave-in-the-public-cloud, 2017.

[148] W. Richter, "Agentless cloud-wide monitoring of virtual disk state," in *Proceedings of the 2014 workshop on PhD forum.* ACM, 2014, pp. 15–16.

[149] H. Zhou, H. Ba, J. Ren, Y. Wang, Y. Li, Y. Chen, and Z. Wang, "Agentless and uniform introspection for various security services in iaas cloud," in *Information Science and Control Engineering (ICISCE), 2017 4th International Conference on.* IEEE, 2017, pp. 140–144.

[150] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation, (OSDI)*, USA, 2006, pp. 307–320.

[151] A. Turk, R. S. Gudimetla, E. U. Kaynar, J. Hennessey, S. Tikale, P. Desnoyers, and O. Krieger, "An experiment on bare-metal bigdata provisioning," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16).* Denver, CO: USENIX Association, Jun. 2016. [Online]. Available: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/turk

[152] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina *et al.*, "Rack-scale disaggregated cloud data centers: The dredbox project vision," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe.* EDA Consortium, 2016, pp. 690–695.

[153] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, "Flash storage disaggregation," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 29:1–29:15. [Online]. Available: http://doi.acm.org/10.1145/2901318.2901337

[154] H. M. M. Ali, A. Q. Lawey, T. E. El-Gorashi, and J. M. Elmirghani, "Energy efficient disaggregated servers for future data centers," in *2015 20th European Conference on Networks and Optical Communications-(NOC)*. IEEE, 2015, pp. 1–6.

[155] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[156] "Federal risk and authorization management program." [Online]. Available: https://www.fedramp.gov

[157] "Payment card industry security standards council." [Online]. Available: https://www.pcisecuritystandards.org

[158] "Center for internet security." [Online]. Available: https://www.cisecurity.org

[159] "Hcl appscan," https://www.hcltechsw.com/wps/portal/products/appscan.

[160] O. Tuncer, N. Bila, S. Duri, C. Isci, and A. K. Coskun, "Confex: Towards automating software configuration analytics in the cloud," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2018, pp. 30–33.

[161] S. Baset, S. Suneja, N. Bila, O. Tuncer, and C. Isci, "Usable declarative configuration specification and validation for applications, systems, and cloud," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track*. ACM, 2017, pp. 29–35.

[162] T. Chiba, R. Nakazawa, H. Horii, S. Suneja, and S. Seelam, "Confadvisor: A performance-centric configuration tuning framework for containers on kubernetes," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2019, pp. 168–178.

[163] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 307–320.

[164] J. Hennessey, S. Tikale, A. Turk, E. U. Kaynar, C. Hill, P. Desnoyers, and O. Krieger, "Hil: designing an exokernel for the data center," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 2016, pp. 155–168.

[165] K. Z. Meth and J. Satran, "Design of the iscsi protocol," in *Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*. IEEE, 2003, pp. 116–122.

[166] H. P. Anvin and M. Connor, "X86 network booting: Integrating gpxe and pxelinux," in *Linux Symposium*. Citeseer, 2008, p. 9.

[167] T. Fujita and M. Christie, "tgt: Framework for storage target drivers," in *Proceedings of the Linux Symposium*, vol. 1. Citeseer, 2006, pp. 303–312.

[168] "Ubuntu Security Notices," https://usn.ubuntu.com/.

[169] "Agentless System Crawler," https://github.com/cloudviz/agentless-system-crawler.

[170] T. Bray, "The javascript object notation (json) data interchange format," Tech. Rep., 2017.

[171] "Open Security Content Automation Protocol," https://www.open-scap.org/tools/openscap-base/.

[172] R. Battle and E. Benson, "Bridging the semantic web and web 2.0 with representational state transfer (rest)," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 6, no. 1, pp. 61–69, 2008.

[173] "Mass Open Cloud," https://massopen.cloud/.

[174] "mysql relational database managament system," https://www.mysql.com/.

[175] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 2009, pp. 44–54.

[176] "Apache HTTP Server Project," https://httpd.apache.org/.

[177] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The NAS parallel

benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[178] "ab - Apache HTTP server benchmarking tool," https://httpd.apache.org/docs/2.4/programs/ab.html.

[179] C. Curino, S. Krishnan, K. Karanasos, S. Rao, G. M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, S. Heddaya, R. Burd, S. Sakalanaga, C. Douglas, B. Ramsey, and R. Ramakrishnan, "Hydra: a federated resource manager for data-center scale analytics," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 177–192. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/curino

[180] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 18.

[181] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: the next generation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–14.

[182] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand *et al.*, "Autopilot: workload autoscaling at google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[183] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi *et al.*, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." in *NSDI*, vol. 11, 2011, pp. 22–22. [Online]. Available: http://static.usenix.org/events/nsdi11/tech/full_papers/Hindman_new.pdf

[184] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand, "Firmament: Fast, centralized cluster scheduling at scale," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 99–115.

[185] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: Scalable and coordinated scheduling for cloud-scale computing," in *11th*

{*USENIX*} *Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 285–300.

[186] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1–16.

[187] D. Aivaliotis, "Tupperware: Container deployment at scale," 2015.

[188] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 261–276.

[189] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu, "Fuxi: A fault-tolerant resource management and job scheduling system at internet scale," *Proc. VLDB Endow.*, vol. 7, no. 13, p. 1393–1404, Aug. 2014. [Online]. Available: https://doi.org/10.14778/2733004.2733012

[190] F. Liu, K. Keahey, P. Riteau, and J. Weissman, "Dynamically negotiating capacity between on-demand and batch clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[191] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *SIGOPS European Conference on Computer Systems (EuroSys)*, Prague, Czech Republic, 2013, pp. 351–364. [Online]. Available: http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf

[192] Q. Liu and Z. Yu, "The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from alibaba trace," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 347–360. [Online]. Available: https://doi.org/10.1145/3267809.3267830

[193] A. O. F. Atya, Z. Qian, S. V. Krishnamurthy, T. La Porta, P. McDaniel, and L. Marvel, "Malicious co-residency on the cloud: Attacks and defense," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[194] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[195] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 973–990.

[196] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[197] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 1–18.

[198] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 199–212.

[199] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "Crosstalk: Speculative data leaks across cores are real."

[200] N. Chakthranont, P. Khunphet, R. Takano, and T. Ikegami, "Exploring the performance impact of virtualization on an hpc cloud," in *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*. IEEE, 2014, pp. 426–432.

[201] N. Kratzke, "About microservices, containers and their underestimated impact on network performance," *arXiv preprint arXiv:1710.04049*, 2017.

[202] P. Lubomski, A. Kalinowski, and H. Krawczyk, "Multi-level virtualization and its impact on system performance in cloud computing," in *International Conference on Computer Networks*. Springer, 2016, pp. 247–259.

[203] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in *2015 IEEE International Conference on Cloud Engineering*. IEEE, 2015, pp. 386–393.

[204] 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)*. IEEE, 2015, pp. 171–172.

[205] Y. Zhou, B. Subramaniam, K. Keahey, and J. Lange, "Comparison of virtualization and containerization techniques for high performance computing," in *Proceedings of the 2015 ACM/IEEE conference on Supercomputing*, 2015.

[206] W.-Y. Chen, K.-J. Ye, C.-Z. Lu, D.-D. Zhou, and C.-Z. Xu, "Interference analysis of co-located container workloads: A perspective from hardware performance counters," *Journal of Computer Science and Technology*, vol. 35, pp. 412–417, 2020.

[207] Red Hat Inc., "Open cloud testbed," https://research.redhat.com/blog/research_project/open-cloud-testbed/, 2019.

[208] National Science Foundation, "Ccri: Grand: Developing a testbed for the research community exploring next-generation cloud platforms," https://www.nsf.gov/awardsearch/showAward?AWD_ID=1925504, 2019.

[209] OPENINFRA LABS, "Operate first community manifesto," https://openinfralabs.org/operate-first-community-manifesto/, 2019.

[210] NSA, "Utah Data Center," 2018. [Online]. Available: https://nsa.gov1.info/utah-data-center/

[211] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 153–167.

[212] G. Amvrosiadis, J. W. Park, G. R. Ganger, G. A. Gibson, E. Baseman, and N. DeBardeleben, "On the diversity of cluster workloads and its impact on research results," in *2018 USENIX Annual Technical Conference (USENIX ATC*

18). Boston, MA: USENIX Association, 2018, pp. 533–546. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/amvrosiadis

[213] C. Delimitrou and C. Kozyrakis, "Hcloud: Resource-efficient provisioning in shared cloud systems," *SIGARCH Comput. Archit. News*, vol. 44, no. 2, p. 473–488, Mar. 2016. [Online]. Available: https://doi.org/10.1145/2980024.2872365

[214] "ATLAS." [Online]. Available: https://atlas.web.cern.ch/Atlas/Collaboration/

[215] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks and ISDN systems*, vol. 17, no. 1, pp. 1–14, 1989.

[216] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *23rd IEEE Int. Symp. on Parallel and Distributed Processing (IPDPS-09)*, 2009, pp. 1–12.

[217] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*, Jan. 2018.

[218] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 605–622.

[219] Y. A. Younis, K. Kifayat, and A. Hussain, "Preventing and detecting cache side-channel attacks in cloud computing," in *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing*, ser. ICC '17. ACM, 2017, pp. 83:1–83:8. [Online]. Available: http://doi.acm.org/10.1145/3018896.3065843

[220] A. O. F. Atya, Z. Qian, S. V. Krishnamurthy, T. L. Porta, P. McDaniel, and L. Marvel, "Malicious co-residency on the cloud: Attacks and defense," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, May 2017, pp. 1–9.

[221] S. T. King and P. M. Chen, "Subvirt: Implementing malware with virtual machines," in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006, pp. 14–pp.

[222] D. Perez-Botero, J. Szefer, and R. B. Lee, "Characterizing hypervisor vulnerabilities in cloud computing servers," in *Proceedings of the 2013 International Workshop on Security in Cloud Computing*, ser. Cloud Computing '13. New York, NY, USA: ACM, 2013, pp. 3–10. [Online]. Available: http://doi.acm.org/10.1145/2484402.2484406

[223] W. K. Sze, A. Srivastava, and R. Sekar, "Hardening OpenStack Cloud Platforms against Compute Node Compromises," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security - ASIA CCS '16*. Xi'an, China: ACM Press, 2016, pp. 341–352. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2897845.2897851

[224] K. Hogan, H. Maleki, R. Rahaeimehr, R. Canetti, M. van Dijk, J. Hennessey, M. Varia, and H. Zhang, "On the universally composable security of openstack," *IACR Cryptology ePrint Archive*, vol. 2018, p. 602, 2018. [Online]. Available: https://eprint.iacr.org/2018/602

[225] Packet, "The promise of the cloud delivered on bare metal," https://www.packet.net, 2017.

[226] A. W. S. Inc., "Amazon EC2 Bare Metal Instances with Direct Access to Hardware," https://aws.amazon.com/blogs/aws/new-amazon-ec2-bare-metal-instances-with-direct-access-to-hardware/, 2017.

[227] "Hil: Hardware Isolation Layer, formerly Hardware as a Service," https://github.com/CCI-MOC/hil.

[228] ANONOMIZED, "Anonomized firmware," MMM YYY.

[229] T. Hudson, "Linuxboot," https://github.com/osresearch/linuxboot.

[230] ANONOMIZED, "Anonomized provisioning service," MMM YYY.

[231] "Malleable Metal as a Service (M2)," https://github.com/CCI-MOC/M2.

[232] N. Schear, P. T. Cable, II, T. M. Moyer, B. Richard, and R. Rudd, "Bootstrapping and maintaining trust in the cloud," in *Proceedings of the 32nd Annual Conference on Computer Security Applications, (SCSAC)*. USA: ACM, 2016, pp. 65–77.

[233] "python-keylime: Bootstrapping and Maintaining Trust in the Cloud," https://github.com/mit-ll/python-keylime.

[234] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold boot attacks on encryption keys," in *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, 2008, pp. 45–60. [Online]. Available: http://www.usenix.org/events/sec08/tech/full_papers/halderman/halderman.pdf

[235] J. Szefer, P. Jamkhedkar, D. Perez-Botero, and R. B. Lee, "Cyber defenses for physical attacks and insider threats in cloud computing," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '14. New York, NY, USA: ACM, 2014, pp. 519–524. [Online]. Available: http://doi.acm.org/10.1145/2590296.2590310

[236] M. Guri, B. Zadov, D. Bykhovsky, and Y. Elovici, "PowerHammer: Exfiltrating Data from Air-Gapped Computers through Power Lines," *arXiv:1804.04014 [cs]*, Apr. 2018, arXiv: 1804.04014. [Online]. Available: http://arxiv.org/abs/1804.04014

[237] "Trusted Platform Module (TPM) Summary," https://trustedcomputinggroup.org/trusted-platform-module-tpm-summary/, Apr. 2008.

[238] T. Hudson and L. Rudolph, "Thunderstrike: EFI firmware bootkits for Apple Macbooks," in *Proceedings of the 8th ACM International Systems and Storage Conference*. ACM, 2015, p. 15.

[239] T. Hudson, X. Kovah, and C. Kallenberg, "ThunderStrike 2: Sith Strike," *Black Hat USA Briefings*, 2015.

[240] H. Wagner, D.-I. M. Zach, and D.-I. F. M. A.-P. Lintenhofer, "BIOS-rootkit LightEater," 2015.

[241] Y. Bulygin, J. Loucaides, A. Furtak, O. Bazhaniuk, and A. Matrosov, "Summary of attacks against BIOS and secure boot," *Defcon-22*, 2014.

[242] J. Rutkowska, "Intel x86 considered harmful," 2015, https://blog.invisiblethings.org/papers/2015/x86_harmful.pdf.

[243] J. Heasman, "Rootkit threats," *Network Security*, vol. 2006, no. 1, pp. 18–19, 2006.

[244] J. Butterworth, C. Kallenberg, X. Kovah, and A. Herzog, "BIOS Chronomancy: Fixing the core root of trust for measurement," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516714

[245] B. Morgan, E. Alata, V. Nicomette, and M. Kaâniche, "Bypassing IOMMU protection against I/O attacks," in *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, Oct 2016, pp. 145–150.

[246] R. Wojtczuk and J. Rutkowska, "Attacking intel trusted execution technology," *Black Hat DC*, 2009.

[247] W. A. Arbaugh, "Trusted computing," *Department of Computer Science, University of Maryland,[online][Retrieved on Feb. 22, 2007] Retrieved from the Internet*, 2007.

[248] A. Regenscheid, "Platform firmware resiliency guidelines," https://doi.org/10.6028/NIST.SP.800-193, May 2018.

[249] "ABOUT THE MGHPCC | MGHPCC," http://www.mghpcc.org/about/about-the-mghpcc/.

[250] "NWRDC | The Ultimate Solution to Simplify Your Data Center," http://www.nwrdc.fsu.edu/.

[251] "Equinix Private Cloud Architecture," https://www.equinix.com/solutions/cloud-infrastructure/private-cloud/architecture/.

[252] D. S. Anderson, M. Hibler, L. Stoller, T. Stack, and J. Lepreau, "Automatic online validation of network configuration in the emulab network testbed," in *Autonomic Computing, 2006. ICAC'06. IEEE International Conference on.* IEEE, 2006, pp. 134–142.

[253] Openstack, "Ironic," https://docs.openstack.org/ironic/latest/, 2018.

[254] "Metal as a service(maas) from canonical," https://maas.io/, 2018.

[255] IEEE Computer Society, *IEEE standard for local and metropolitan area networks media access control (MAC) bridges and virtual bridged local area networks.* New York: Institute of Electrical and Electronics Engineers, 2018. [Online]. Available: https://standards.ieee.org/standard/802_1Q-2018.html

[256] "What is TianoCore?" https://www.tianocore.org/.

[257] "Coreboot minimal firmware," https://doc.coreboot.org/.

[258] "coreboot - payloads," https://doc.coreboot.org/payloads.html.

[259] "Linux unified key setup," https://gitlab.com/cryptsetup/cryptsetup/blob/master/README.md, 2018.

[260] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a tcg-based integrity measurement architecture," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 16–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251375.1251391

[261] F. Tomonori and M. Christie, "tgt: Framework for storage target drivers," in *Linux Symposium*, 2006.

[262] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation.* USENIX Association, 2006, pp. 307–320.

[263] P. Z. Gal Beniamini, "Over the air: Exploiting Broadcom's wi-fi stack," https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html.

[264] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell, "Linux kernel integrity measurement using contextual inspection," in *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, ser. STC '07. New York, NY, USA: ACM, 2007, pp. 21–29. [Online]. Available: http://doi.acm.org/10.1145/1314354.1314362

[265] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM*

Comput. Surv., vol. 50, no. 1, pp. 16:1–16:33, Apr. 2017. [Online]. Available: http://doi.acm.org/10.1145/3054924

[266] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 268–279. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813691

[267] IBM, "Ibm's tpm 1.2," http://ibmswtpm.sourceforge.net/, 2019.

[268] "Strongswan," https://www.strongswan.org/, Oct. 2018.

[269] A. Hoban, "Using intel® aes new instructions and pclmulqdq to significantly improve ipsec performance on linux," https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/aes-ipsec-performance-linux-paper.pdf, August 2010.

[270] Foreman, "Foreman," https://www.theforeman.org/, 2019.

[271] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, ser. HotCloud '10, 2010.

[272] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 153–167. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132772

[273] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," https://github.com/filebench/filebench/wiki, 2017.

[274] R. Ricci and t. E. Team, "Precursors: Emulab," in *The GENI Book*, R. McGeer, M. Berman, C. Elliott, and R. Ricci, Eds. Cham: Springer International Publishing, 2016, pp. 19–33. [Online]. Available: https://doi.org/10.1007/978-3-319-33769-2_2

[275] IBM, "Extreme Cloud Administration Toolkit — xCAT 2.14.5 documentation," https://xcat-docs.readthedocs.io/en/stable/index.html#, 2019.

[276] A. Mosayyebzadeh, G. Ravago, A. Mohan, A. Raza, S. Tikale, N. Schear, T. Hudson, J. Hennessey, N. Ansari, K. Hogan, C. Munson, L. Rudolph, G. Cooperman, P. Desnoyers, and O. Krieger, "A secure cloud with minimal provider trust," in *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. Boston, MA: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/hotcloud18/presentation/mosayyebzadeh

[277] "Titan in depth: Security in plaintext," https://cloud.google.com/blog/products/gcp/titan-in-depth-security-in-plaintext/, 2019.

[278] "Project Cerberus Architecture Overview," https://github.com/opencomputeproject/Project_Olympus/tree/master/Project_Cerberus, Dec 2018.

[279] T. Fukai, S. Takekoshi, K. Azuma, T. Shinagawa, and K. Kato, "BMCArmor: A Hardware Protection Scheme for Bare-Metal Clouds," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec 2017, pp. 322–330.

[280] IBMcloud, "Hardware monitoring and security controls," https://console.bluemix.net/docs/bare-metal/intel-trusted-execution-technology-txt.html#hardware-monitoring-and-security-controls, Apr 2018.

[281] O. Inc, "Oracle Cloud Infrastructure Security," *Oracle Cloud Infrastructure white papers*, p. 36, Nov 2018.

[282] X. Kovah, C. Kallenberg, J. Butterworth, and S. Cornwell, "SENTER Sandman: Using Intel TXT to Attack BIOSes," in *HITB Security Conference*, Amsterdam, May 2014, p. 5. [Online]. Available: http://conference.hitb.org/hitbsecconf2014kul/wp-content/uploads/2014/08/HITB2014KUL-SENTER-Sandman.pdf

[283] H. Moore, "A penetration tester's guide to ipmi and bmcs," https://blog.rapid7.com/2013/07/02/a-penetration-testers-guide-to-ipmi/, Aug 2017.

[284] L. H. Newman, "Intel chip flaws leave millions of devices exposed," https://www.wired.com/story/intel-management-engine-vulnerabilities-pcs-servers-iot/, Nov 2017.

[285] M. Ermolov and M. Goryachy, "How to hack a turned -off computer, or running unsigned code in intel manage-

ment engine," https://www.blackhat.com/docs/eu-17/materials/eu-17-Goryachy-How-To-Hack-A-Turned-Off-Computer-Or-Running-Unsigned-Code-In-Intel-Manage pdf, Dec 2017.

[286] A. Kroizer, "Tpm and intel ® ptt overview," http://tce.webee.eedev.technion.ac.il/wp-content/uploads/sites/8/2016/01/AK_TPM-overview-technion.pdf, Sep 2015.

[287] J. Kirk, "Destroying your hard drive is the only way to stop this super-advanced malware," https://www.pcworld.com/article/2884952/equation-cyberspies-use-unrivaled-nsastyle-techniques-to-hit-iran-russia.html, Feb 2015.

[288] "me_cleaner: Tool for partial deblobbing of intel me/txe firmware images," https://github.com/corna/me_cleaner, 2018.

Appendix A

Supporting Security Sensitive Tenants in a Bare-Metal Cloud

A.1 Introduction

There are a number of security concerns with today's clouds. First, virtualized clouds collocate multiple tenants on a single physical node, enabling malicious tenants to launch side-channel and covert channel attacks [198, 217, 218, 197, 219, 220] or exploit vulnerabilities in the hypervisor to launch attacks both on tenants running on the same node [221, 222] and on the cloud provider itself[223]. Second, popular cloud management software like OpenStack can have a trusted computing base (TCB) with millions of lines of code and a massive attack surface[224]. Third, for operational efficiency, cloud providers tend to support *one-size-fits-all* solutions, where they apply uniform solutions (e.g. network encryption) to all customers; meeting the specialized requirements of highly security sensitive customers may impose unacceptable costs for others. Finally, and perhaps most concerning, existing clouds provide tenants with very limited visibility and control over internal operations and implementations; the tenant needs to fully trust the non-maliciousness and competence of the provider.

While bare-metal clouds [110, 111, 113, 225, 226] eliminate the security concerns implicit in virtualization, they do not address the rest of the challenges described above. For example, OpenStack's bare-metal service still has all of OpenStack in the TCB. As another example, existing bare-metal clouds ensure that previous tenants have not compromised

firmware by adopting a one-size-fits-all approach of validation/attestation or re-flashing firmware. The tenant has no way to programmatically verify the firmware installed and needs to fully trust the provider. As yet another example, existing bare-metal clouds require the tenant to trust the provider to scrub any persistent state on the physical machine before allocating the machine to other tenants.[1]

These issues are a major concern for "security-sensitive" organizations, which we define as entities that are both willing to pay a significant price (dollars and/or performance) for security and that have the expertise, desire, or requirement to trust their own security arrangements over those of a cloud provider. Many medical, financial and federal institutions fit into this category. Recently, IARPA, who represents a number of such entities, released an RFI [147] that describes their requirement for using future public clouds; to "*replicate as closely as possible the properties of an air-gapped private enclave*" of physical machines. More concretely[2] this means a cloud where the tenant trusts the provider to make systems available but where confidentiality and integrity for a tenant's *enclave* is under the control of the tenant who is free to implement their own specialized security processes and procedures.

By our definition the majority of computing demands are not highly security-sensitive, thus providing a high-security option within a commercially-viable future cloud must not impact the efficiency of providing service to other tenants. Is this possible? Can we make a cloud that is appropriate for even the most security sensitive tenants? Can we make a cloud where the tenant does not need to fully trust the provider? Can we do this without performance impact on tenants who are happy with the security levels of today's clouds?

The Bolted architecture and prototype implementation, described in this work, demonstrates that the answer to these questions is "yes." The fundamental insight is that to implement a bare metal cloud only a minimum *isolation service* need to be controlled by the provider; all other functionality can be implemented by security-sensitive tenants on their own behalf, with provider-maintained implementations available to tenants with more typical security needs.

Bolted defines a set of micro-services, namely an *isolation service* that uses network isolation technologies to isolate tenants' bare-metal servers, a *provisioning service* that installs software on servers using network mounted storage, and an *attestation service* that

[1] See, for example, IBM Cloud's security policy for scrubbing local drives here https://tinyurl.com/y75sakn4. Note that scrubbing local disks can require hours of overhead on transferring computers between tenants; dramatically impacting the elasticity of the cloud.

[2] Per private communications with RFI authors.

compares measurements (hashes) of firmware/software on a server against a whitelist of allowed software. All services can be deployed by the provider as a one-size-fits-all solution for the tenants that are willing to trust the provider.

Security sensitive tenants can deploy their own provisioning and attestation service thereby minimizing their trust in the provider. The tenant's own software executing on machines (already trusted by the tenant), can validate measurements of code to be executed on some newly allocated server against her expectations rather than having to trust the provider. Further, the tenant's attestation service can securely distribute keys to the server for network and disk encryption. Using the default implementation of Bolted services, a tenant's enclave is protected from previous users of the same servers (using hardware-based attestation), from concurrent tenants of the cloud (using network isolation and encryption), and from future users of the same servers (using network mounted storage, storage encryption, and memory scrubbing). Further, a tenant with specialized needs can modify these services to match their requirements; the provider does not sacrifice operational efficiency or flexibility for security-sensitive customers with specialized needs since it is the tenant and not the provider responsible for implementing complex policies.

Key contributions of this work are:

An **architecture** for a bare-metal cloud that: 1) enables security-sensitive tenants to control their own security while only trusting the provider for physical security and availability while 2) not imposing overhead on tenants that are security insensitive and not compromising the flexibility or operational efficiency of the provider. Key elements of the architecture are: 1) disk-less provisioning that eliminates the need to trust the provider for disk scrubbing (as well as the huge cost), 2) remote attestation (versus validation or re-flashing) to provide the tenant with a proof of the firmware and software running on their server and 3) secure deterministically built firmware that allows the tenant to inspect the source code used to generate the firmware.

A **prototype implementation** of the Bolted architecture where all its components are made available by us open-source, including the isolation service (Hardware Isolation Layer [117, 227]), a deterministic Linux-based minimal firmware (LinuxBoot [228, 229], a disk-less bare-metal provisioning service (Bare Metal Imaging [230, 231]), a remote attestation service (Keylime [232, 233]), and scripts that interact with the various services to elastically create secure private enclaves. As we will discuss later, only the microservice providing isolation (i.e., Hardware Isolation Layer) is in the TCB and we show that this can, in fact,

be quite small; just over 3K LOC in our implementation.

A **performance evaluation** of the Bolted prototype that demonstrates: 1) elasticity similar to today's virtualized cloud (∼3 minutes to allocate and provision a physical server), 2) the cost of attestation has a modest impact ∼25% on the provisioning time, 3) there is value for customers that trust the provider in avoiding extra security (e.g.,∼200% for some applications), while 4) security-sensitive customers can still run many non-IO intensive applications with negligible overhead and even I/O intensive BigData applications with a relatively modest (e.g., ∼30%) degradation.

A.2 Threat Model

We describe the threats to the victim, a tenant renting bare-metal servers from the cloud, and describe approaches taken by Bolted to safeguards against them. We consider external entities (hackers), malicious insiders in the cloud provider's organization and all other tenants of the server—both past and future—as potential adversaries to the victim. We assume that the goal of the adversary is to steal data, corrupt data, or deny services to the victim by gaining access to the victim's occupied servers or network. Our goal is to empower the tenant with the ability to take control of its own security; it is up to the tenant to make the tradeoff decision between the degree to which it relies on the provider's security systems versus the harm that it may suffer from a successful attack.

The cloud provider is always trusted with the physical security of the datacenter, thus any attacks involving physical access to the infrastructure, including power and noise analysis, bus snooping, or decapping chips [234, 235, 236] are out of scope of a tenant's control. The provider is also trusted for the availability of the network, node allocation services, and any network performance guarantees. We assume that the cloud itself is vulnerable to exploitation by external entities (hackers) or a malicious insider (e.g., a rogue systems administrator) but we trust the cloud provider's organization to have necessary systems and procedures in place to detect and limit the impact of such events. For example, the provider can enforce sufficient technical separation of duties (e.g., two-person rule) such that a single malicious insider or hacker cannot both re-flash all the node firmware in a data center and change what hashes the provider publishes for attestation, have both physical and logical access to a node, or make unreviewed changes to the provider's deployed software, etc. Further, we assume that all servers in the cloud are equipped with a Trusted Platform Module

(TPM) - a dedicated cryptographic coprocessor required for hardware-based authentication [237].

We categorize the threats that the tenant faces into the following phases:

Prior to occupancy: Malicious (or buggy) firmware can threaten the integrity of a server, as well as that of other servers it is able to contact. A tenant server's firmware may be infected prior to the tenant using it, either by the previous tenant (e.g., by exploiting firmware bugs) or by the cloud provider insider (e.g., by unauthorized firmware modification). If a server is not sufficiently isolated from potential attackers there is also a threat of infection between the time it is booted until it is fully provisioned and all defenses are in place.

During occupancy: Although many side-channel attacks are avoided by disallowing concurrent tenants on the same server, if the server's network traffic is not sufficiently isolated, the provider or other concurrent tenants of the cloud may be able to launch attacks against it or eavesdrop on its communication with other servers in the enclave. Moreover, if network attached storage is used (as in our implementation) all communication that is not sufficiently secured between server and storage may be vulnerable. Finally, there is a threat to the tenant from denial of service attacks.

After occupancy: Once the tenant releases a server, the confidentiality of a tenant may be compromised by any of its state (e.g, storage or memory) being visible to subsequent software running on the server.

A.3 Design Philosophy

The key goals of Bolted are: (1) to minimize the trust that a tenant needs to place in the provider, (2) to enable tenants with specialized security expertise to implement the functionality themselves, and (3) to enable tenants to make their own cost/performance/security tradeoffs – in bare-metal clouds. These goals have a number of implications in the design of Bolted.

First, Bolted differs from existing bare metal offerings in that most of the component services that make up Bolted can be operated by a tenant rather than by the provider. A security sensitive tenant can customize or replace these services. All the logic that orchestrates how different services are used to securely deploy a tenant's software is implemented using scripts that can be replaced or modified by the user. Most importantly, the service that

checks the integrity of a rented server can be deployed (and potentially re-implemented) by the tenant.

Second, while we expect a provider to secure and isolate the network and storage of tenants, we only rely on the provider for availability and not for the confidentiality or integrity of the tenant's computation. For tenants that do not trust the provider, we assume that Bolted tenants will further encrypt all communication between the their servers and between those servers and storage. Bolted provides a (user-operated) service to securely distribute keys for this purpose.

Third, we rely on attestation (measuring all firmware and software and ensuring that it matches known good values) that can be implemented by the tenant rather than just validation (ensuring that software/firmware is signed by a trusted party). This is critical for firmware which may contain bugs [238, 239, 240, 241, 242, 243] that can disrupt tenant security. Attestation provides a time-of-use proof that the provider has kept the firmware up to date. More generally, the whole process of incorporating a server into an enclave can be attested to the tenant. In addition, the tenant can continuously attest when the server is operating, ensuring that any code loaded in any layer of software (OS, applications and etc., and irrespective of who signed them) can be dynamically checked against a tenant-controlled whitelist.

Fourth, we have a strong focus on keeping our software as small as possible and making it all available via open source. In some cases, we have written our own highly specialized functionality rather than relying on larger function rich general purpose code in order to achieve this goal. For functionality deployed by the provider, this is critical to enable it to be inspected by tenants to ensure that any requirements are met. For example, previous attacks have shown that firmware security features are difficult to implement bug-free – including firmware measurements being insufficient [244], hardware protections against malicious devices not being in place [245], and dynamic root of trust (DRTM) implementation flaws [246]. Further, our firmware is deterministically built, so that the tenant can not only inspect it for correct implementation but then easily check that this is the firmware that is actually executing on the machine assigned to the tenant. For tenant-deployed functionality, small open source implementations are valuable to enable user-specific customization.

Finally, servers are assumed to be stateless with all volumes accessed on-demand over the network. This removes confidentiality or denial of service attacks by the provider or

subsequent tenants of server inspecting or deleting a tenants disk state. Bare-metal clouds that support stateful servers need to either give the tenant the guarantee that a node will never be preempted (problematic in a pay-for-use cloud model) or ensure that the provider scrubs the disks (trusting the provider and potentially requiring hours with modern disks). As we will see, stateless servers also dramatically improve the elasticity of the service.

A.4 Architecture

Bolted enables tenants to build a secure enclave of bare-metal servers where the integrity of each server is verified by the tenant before it is allowed to participate in the tenant's enclave. During the allocation process, a server transitions through the following states: **free**, or not allocated, **airlock**, where the integrity of the server is checked, after which it is either **allocated** to a tenant's secure enclave if it passes the integrity check or **rejected** if it fails. In this section, we discuss the Bolted components; their operations; the process of server allocation, attestation, and the degrees of freedom in deploying Bolted components to support different security requirements and use cases.

A.4.1 Components

Bolted consists of four components which operate independently and (in the highest-security and lowest-trust configurations) are orchestrated by the tenant rather than the provider.

Isolation Service: The Isolation Service exposes interfaces to (de)allocate servers and networks to tenants, and isolate and/or group the servers by manipulating a provider's networking infrastructure (switches and/or routers). Using the exposed interfaces, the servers are moved to *free* or *rejected* state as well – ensuring the servers are not part of any tenant-owned network. These interfaces are also used to move the servers to the *airlock* state (to verify if they have been compromised) or the *allocated* state (where they are available for the tenant).

The Isolation Service uses network isolation techniques instead of encryption-based logical isolation in order to enforce guarantees of performance and to provide basic protection against traffic analysis attacks. Since the operations performed by these interfaces (on the networking infrastructure) are privileged, the isolation service needs to be deployed by the

provider; if a tenant does not trust the provider, it can further encrypt network traffic between their servers.

Secure Firmware: Secure firmware is crucial towards improving tenant's trust of the public cloud servers; it should consist of following properties. First, it should be open-source, so that it benefits from large community support in improving its features and fixing any bugs and vulnerabilities. Second, it should be deterministically built so that a tenant can build the firmware from verified source code and independently validate the provider-installed firmware. Third, it must scrub server memory prior to launching a tenant OS – if the server was preempted from a previous tenant, it must guarantee that the previous tenants' code and data is not present in the memory. Finally, it must provide an execution environment for the attestation agent in the airlock state.

We note that it is challenging to replace computer firmware; even major providers are often forced to install huge binary blobs signed by the hardware manufacturer with no access to the source code. When firmware cannot be replaced, we use the installed firmware for the minimum amount of time in order to download our own secure firmware – and the servers' pre-installed firmware must support trusted boot [247].

While the overall Bolted architecture design supports the attestation and security of both system firmware (e.g., BIOS or UEFI) and peripheral firmware (e.g., GPU, network card, etc.), there are no standardized and implemented methods to attest those peripheral firmware to an external party. Early attempts at standardization are underway, and we expect Bolted can leverage them when they mature [248].

Provisioning Service: This service is broadly responsible for three things – (1) initial provisioning of the server with the software stack (i.e. secure firmware and attestation agent) responsible for its attestation during the *airlock* state, (2) provisioning of the server during the *allocated* state (i.e. the server was successfully verified that it was not compromised) with the intended software stack i.e. the operating system and the relevant software packages, and (3) saving and/or deleting the servers' persistent state when a server is released.

The Provisioning Service can be deployed either by the provider or by tenants themselves. The latter option is valuable for security-sensitive tenants who do not want to trust the provider with their operating system images or who want to use their own (e.g., legacy) provisioning systems. The provisioning service must provision the servers in a stateless manner so that the tenants do not have to rely on (and trust) the provider to

remove any persistent state after the server is released.

Attestation Service: The Attestation Service consists of two parts: an attestation agent that executes on the server to be attested, and an attestation server that maintains a pre-populated database of known reliable hash measurements of allowed firmware/software (i.e., a whitelist). This service is used during the *airlock* and *allocated* states. The Attestation Service can be deployed either by the provider or by the tenant.

During the *airlock* state, the attestation agent (downloaded from the Provisioning Service during initial provisioning) is responsible for sending *quotes*[3] of the firmware and any other software involved during the boot sequence to the attestation server to be matched against the whitelist. Depending on the attestation result obtained from the attestation server, the state of the attested server is changed to allocated or rejected. In the case when the computer firmware cannot be replaced, the trusted boot sequence measurement (until the secure firmware is executed) must be supplied by the provider. Obtaining this measurement is a one-time operation for each server, and this whitelist can be published publicly by the provider.

In the *allocated* state, the attestation agent (installed on the tenants' OS) can continuously verify the software stack running against the whitelist present on the attestation server (also referred as *Continuous Attestation*). For continuous attestation to work, the software stack should be configured such that it saves new measurements to the cryptoprocessor upon observing any change/modification/access. The attestation agent periodically sends the new hash measurements of software and configuration registered in the cryptoprocessor to the attestation server; if attestation fails (i.e., when any malicious activity is observed), the attestation server alerts the attestation agent. Continuous attestation protects tenants both against unauthorized execution of executables and against malicious reboots into unauthorized firmware, bootloader, or operating system. Note that continuous attestation is fundamentally more challenging in a provider-deployed attestation service, as the runtime whitelist (e.g., hashes of approved binaries allowed to be run on the node) must be tenant-generated; we assume continuous attestation is only used by security-sensitive tenants that deploy their own attestation service.

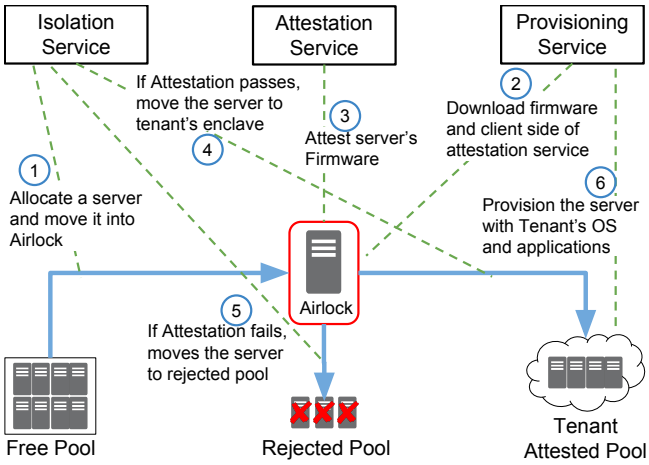


Figure A.1: Bolted's Architecture: Blue arrows show state changes and green dotted lines shows the actions during a state change.

A.4.2 Life Cycle

The different Bolted components do not directly interact with each other, but instead, are orchestrated by user-controlled scripts. Figure A.1 shows the **life-cycle of a typical secure server** (in the case of security-sensitive tenant), which progresses through six steps: (**1**) The tenant uses the Isolation Service to allocate a new bare metal server, create an airlock network, and move the server to that airlock, shared with the Attestation and the Provisioning networks; we need to isolate servers in the airlock state from other servers in the same state so that a compromised server cannot infect other un-compromised servers. (**2**) The secure firmware is executed (if stored in system flash) or provisioned onto the server along with a boot-loader, attestation software agent, and any other related software. With these in place, (**3**) the Attestation Service attests the integrity of the firmware of this server. Once initial attestation completes, (**4**) the tenant again employs the Isolation Service to move the server from the airlock network. If firmware attestation failed (**5**) it is moved into the Rejected Pool, isolated from the rest of the cloud; if attestation was successful, the server is made part of the tenant's enclave by connecting it to the tenant networks. In order to make use of the server, further provisioning is required (**6**) so the tenant again uses the Provisioning Service to install the tenant operating system and any other required applications.

[3]Hash measurements obtained from and signed by a secure cryptoprocessor such as TPM.

A.4.3 Use Cases

Figure A.2 demonstrates the flexibility of Bolted using three examples of users, namely; 1) Alice, a graduate student, who wants to maximize performance and minimize cost and does not care about security, 2) Bob, a professor, who does not trust other tenants (e.g., graduate students) but is willing to trust the provider, and 3) Charlie, a security-sensitive tenant, who not only does not trust other tenants but wants to minimize his trust in the provider.

Alice and Bob are willing to trust the provider's network isolation and storage security, and do not need to employ runtime encryption and will not incur its performance burden; nor will they need to expend the effort to deploy and manage their own services[4]. Alice, further, uses scripts that do not even bother using the provider's attestation service, further improving the speed that she can start up servers as well as her costs if the provider charges her for all the time a server is allocated to her.

Security-sensitive tenant Charlie deploys his own, potentially modified, provisioning and attestation service. He does not have to rely on the provider's network isolation to protect his confidentiality and integrity but can implement runtime protections such as network and disk encryption. Moreover, the attestation service can be used not only to protect him from previous tenants, but also to maintain a whitelist of applications and configuration, and to quickly detect any compromises in an ongoing fashion. The one area where Bolted requires Charlie to trust the provider is for protecting against denial of service attacks since only the provider can deploy the isolation service that allocates servers and controls provider switches. Trusting a provider, in this case, is unavoidable with current networking technology, as the provider controls all networking to the datacenter.

In addition to the cloud use cases, Bolted was designed to be flexible enough to handle the use case of co-location facilities [249, 250, 251] where the datacenter tenants temporarily "loan" computers to each other to handle fluctuations in demand; and this use case is, in fact, the primary one for which Bolted is going into production currently. In this case, a single party may be both provider and tenant. As an example, one party might have a high demand on their HPC cluster, while another party has spare capacity in their IaaS cloud; the isolation service from the second party (the provider) could be used to provision servers for loan to the first party, with attestation and provisioning services

[4]Or mismanage, a more significant risk for less security-literate users.

(including provisioning-associated storage) provided by the first party (the user).

Since the different Bolted services are independent, being orchestrated by tenant scripts, it is straightforward for a tenant to use capacity from multiple isolation services. The attestation of Bolted is important to enable supporting untrusted environments (e.g., research testbeds) alongside production services. For tenants that use the standard Bolted provisioning service, the use of network mounted storage by Bolted enables them to use their own storage for persistence, making storage encryption unnecessary. Because Bolted enables tenants to deploy their own provisioning service, some tenants can use custom provisioning services which install to local storage.[5] When using their own infrastructure, the tenant and provider are in the same organization. In this case, tenants trust the provider, and hence network encryption is unnecessary. Tenants are willing to make agreements with trusted partners from whom they will be using servers; trusting the partner's isolation service makes network encryption unnecessary for communication with servers obtained from it.

A.5 Implementation

We describe our implementation of the Isolation Service (HIL [117]), Firmware (LinuxBoot [228]), Attestation Service (Keylime [232]), and Provisioning Service (BMI [230]), and explain how they work together as Bolted. All of these constituent services of Bolted are open-source packages and can be modified by tenants or providers to meet their specific requirements.

Hardware Isolation Layer: The fundamental operations Hardware Isolation Layer (HIL) provides are (i) allocation of physical servers, (ii) allocation of networks, and (iii) connecting these servers and networks. A tenant can invoke HIL to allocate servers to an enclave, create a management network between the servers, and then connect this network to any provisioning tool (e.g., [252, 253, 230, 254]). It can also let tenants create networks for isolated communication between servers and/or attach those servers to public networks made available by the provider. HIL controls the network switches of the cloud provider and provides VLAN-based [255] network isolation mechanism. HIL also supports a simple API for Baseboard Management Controller (BMC) operations like power cycling servers and console access; ensuring that users cannot attack the BMC. HIL cannot be deployed by tenants and must be deployed by the provider and is the only component shared by tenants,

[5] In this case, provisioning time is much larger and tenants are responsible for scrubbing the local disk.

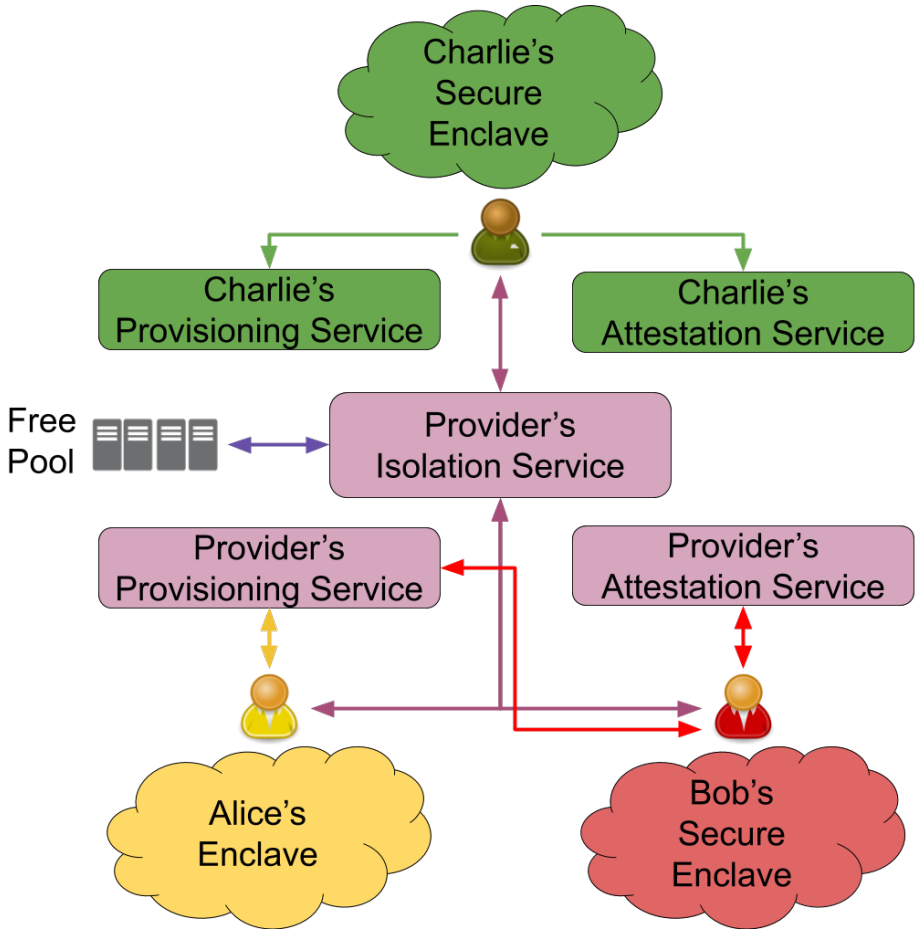


Figure A.2: Bolted deployment examples; purple boxes are provider-deployed and greens are tenant-deployed. Alice and Bob trust the provider-deployed infrastructure, while security-sensitive Charlie deploys its own.

that is not attested to. In our effort to minimize this TCB we have worked hard to keep HIL very simple (approximately 3000 LOC).

Because the provider is trusted for physical isolation and security, it also acts as the source of truth for information on servers in two ways. First, it maps each server's HIL identity to a TPM identity by exporting the TPM's public Endorsement Key (EK) through administrator-modifiable metadata per server, ensuring that the tenant is able to confirm that the tenant she received is indeed the one she reserved thus protecting the tenant from any server spoofing attack. Second, HIL exposes the provider-generated whitelist of TPM PCR measurements, i.e., ones that relate to the platform components like BIOS/UEFI firmware and firmware settings.

LinuxBoot: LinuxBoot is our firmware implementation and bootloader replacement. It is a minimal reproducible build of Linux that serves as an alternative to UEFI and Legacy BIOS. LinuxBoot retains the vendor PEI (Pre-EFI environment) code as well as the signed ACM (authenticated code modules) that Intel provides for establishing the TEE (trusted execution environment). LinuxBoot replaces the DXE (Driver Execution Environment) portion of UEFI with open source wrappers, the Linux Kernel, and a flexible initrd based runtime. Advantages over stock UEFI include: 1) LinuxBoot's open-source Linux devices drivers and filesystems have had significantly more scrutiny than the UEFI implementations, 2) its deterministic build enables easy remote attestation with a TPM; a tenant can independently confirm that the firmware on a server corresponds to source code that they compile themselves, 3) it can use any Linux-supported filesystem or device driver, execute Linux shell scripts to perform remote attestation over secure network protocols and mount encrypted drives, simplifying integration into services like Bolted, 4) it is significantly faster to POST than UEFI; taking 40 seconds on our servers, compared to about 4 minutes with UEFI.

We chose LinuxBoot over alternatives like Tianocore [256] – an open source implementation of UEFI because unlike Tianocore it does not depend on hardware drivers provided by motherboard vendors. In addition to the driver dependency Tianocore also needs support of Firmware Support Package (FSP) from processor vendors which are closed source binaries or independent softwares like coreboot [257, 258] to function as a complete bootable firmware. LinuxBoot does use FSP however Heads which is our flavor of LinuxBoot is able to establish root of trust prior to executing FSP thus ensuring that FSP blob is measured into TPM PCR's. This protects from attacks that involve replacing a measured FSP with a malicious FSP. Additionally, while LinuxBoot and Tianocore both are open source projects, LinuxBoot is based on Linux, a much more mature and widely used system with battle tested code.

We have modified LinuxBoot such that it scrubs memory before a tenant can use a server; a tenant that attests that LinuxBoot is installed is guaranteed that subsequent tenants will not gain control until the memory has been scrubbed since the only way for the provider, or another tenant, to gain control (or reflash the firmware) is to power cycle the machine which will ensure that LinuxBoot is executed. Scripts integrated with LinuxBoot download the attestation service's client side agent, download and kexec a tenant's kernel (only if attestation has succeeded), and obtain a key from the attestation service to access

the encrypted disk and network.

Keylime: Keylime is our remote attestation and key management system. It is divided into four major components: Registrar, Cloud Verifier, Agent, and Tenant. The registrar stores and certifies the public *Attestation Identity Keys (AIKs)* of the TPMs used by a tenant; it is only a trust root and does not store any tenant secrets. The Cloud Verifier (CV) maintains the whitelist of trusted code and checks server integrity. The Agent is downloaded and measured by the server (firmware or previously measured software) and then passes quotes (i.e., TPM-signed attestations of the integrity state of the machine) from the server's TPM to the verifier. The Tenant starts the attestation process and asks the Verifier to verify the server. The Registrar Verifier and Tenant can be hosted by the tenant outside of the cloud or could be hosted on a physical system in the cloud. Keylime delivers the tenant kernel, initrd and scripts to the server (after attestation success) using a secure connection between the Keylime CV and Keylime agent. The script is executed by the agent to 1) make sure the server is on the tenant's secure network and 2) kexec into tenant's kernel and boot the server.

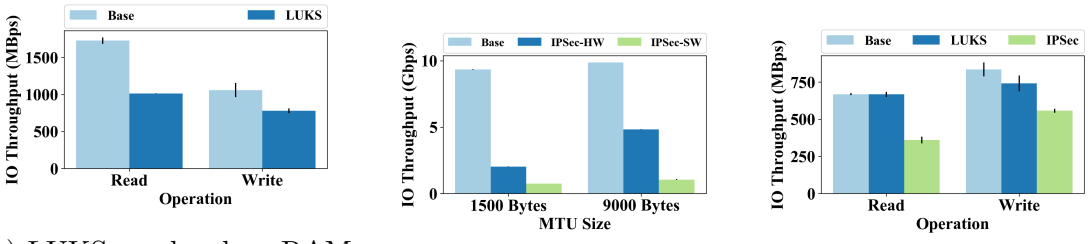
For tenants that do not trust the provider, Keylime supports automatic configuration for Linux Unified Key Setup (LUKS) [259] for disk encryption and IPsec for network encryption using keys bootstrapped during attestation and bound to the TPM hardware root-of-trust. Also, Keylime integrates with the Linux Integrity Measurement Architecture (IMA) [260] to allow tenants to continuously attest that a server was not compromised after boot. IMA continuously maintains a hash chain rooted in the TPM of all programs, libraries, and critical configuration files that have been executed or read by the system. The CV checks the IMA hash chain regularly at runtime to detect deviations from the whitelist of acceptable hashes.

Bare Metal Imaging: The fundamental operations provided by the Bare Metal Imaging (BMI) are: (i) disk image creation, (ii) image clone and snapshot, (iii) image deletion, and (iv) server boot from a specified image. Similar to virtualized cloud services, BMI serves images from remote-mounted boot drives, with server access via an iSCSI (TGT [261]) service managed by BMI and back-end storage in a Ceph [262] distributed storage system. When the server network-boots, it only fetches the parts of the image it uses (less than 1% of the image is typically used), which significantly reduces the provisioning time [230]. BMI allows tenants to run scripts against a BMI-managed filesystem which we use to extract boot information (kernel, initramfs image and kernel command lines) from

images so that they could be passed to a booting server in a secure way via Keylime.

Putting it together: The booting of a server is controlled by a Python application that follows the sequence of steps shown in Figure A.1. For servers that support it, we burn LinuxBoot directly into the server's SPI flash. Figure A.1 shows another case where we download LinuxBoot's runtime (Heads) using iPXE and then continue the same sequence as if LinuxBoot was burned into the flash. We have modified the iPXE client code to measure the downloaded LinuxBoot runtime image into a TPM platform configuration register (PCR) so that all software involved in booting a server can be attested. When servers pass attestation, the Keylime Agent downloads an encrypted zip file containing the tenant's kernel, initrd, and a script from Keylime server and unzips them. The zip file also includes the keys for decrypting the storage and network. After a server is moved (using HIL) to the tenant's enclave, the Keylime Agent runs the script file. The script stores the cryptographic keys into an initrd file to pass it to the tenant's kernel and then kexecs into the downloaded kernel. After it boots, the kernel uses the keys from the initrd file to decrypt the remote disk and encrypt the network.

Keylime [232] and LinuxBoot [228] were previously created in part by authors of this work, and modified as discussed above. While previously published, HIL [117] and BMI [230] were designed with the vision of integrating them in the larger Bolted architecture described in this work.



(a) LUKS overhead on RAM disk

(b) IPsec overhead

(c) Network mounted storage

Figure A.3: Performance Impact of Encryption

A.6 Addressing the Threat Model

Here we discuss how, for security-sensitive tenants, Bolted's architecture addresses the threats in the three phases described in Section A.2.

Prior to occupancy: We must protect a tenant's server against threats from previous users of the server and isolate it from potential network-based attacks until a server is fully provisioned. To do this, Bolted uses attestation to ensure that the firmware of the server was not modified by previous tenants, and isolates the server in the airlock state (protected from other tenants) until this attestation is complete. The deterministic nature of LinuxBoot enables tenants to inspect the source code of the firmware, and ensure that it is trusted, rather than just trusting the provider. Further, all communication within the networks in the enclave is encrypted, using a key provided by the tenant's attestation service (e.g., Keylime), ensuring that the server will not be susceptible to attacks by other servers as it is provisioned. Since our current implementation is unable to attest the state of peripheral firmware, there could be malware embedded in those devices that could compromise the node. Disk and network encryption securely bootstrapped by the TPM mitigate data confidentiality and integrity attacks from malicious peripherals with external access like network interfaces and storage controllers. System level isolation of device drivers, as in Qubes[6], could further be used to mitigate the impact of malicious peripherals mounting attacks against the node [263].

During occupancy: We must ensure that the server's network traffic is isolated so that the provider or other concurrent tenants of the cloud cannot launch attacks against it or eavesdrop on its communication with other servers in the enclave. HIL performs basic VLAN-based isolation to provide basic protection from traffic analysis by other tenants. However, a tenant can choose to both encrypt their network traffic with IPsec and shape their traffic to resist traffic analysis from the provider and not rely on provider's HIL. Keylime securely sends the keys for encrypting networking and disk traffic directly to the node. Disk encryption ensures the confidentiality and integrity of the persistent data even if the storage is under the control of a malicious provider.

Continuous attestation can detect changes to the runtime state of the server (e.g., unauthorized binaries being executed or reboot to an unauthorized kernel) and notify the tenant to take some action to respond. Response actions include revoking the cryptographic keys used by that server for network/storage encryption, removing it from the enclave VLAN, and immediately rebooting the system into a known good state and scrubbing its memory. While IMA only supports load/read-time measurement (i.e., hashing) of files on

[6] `https://www.qubes-os.org/`

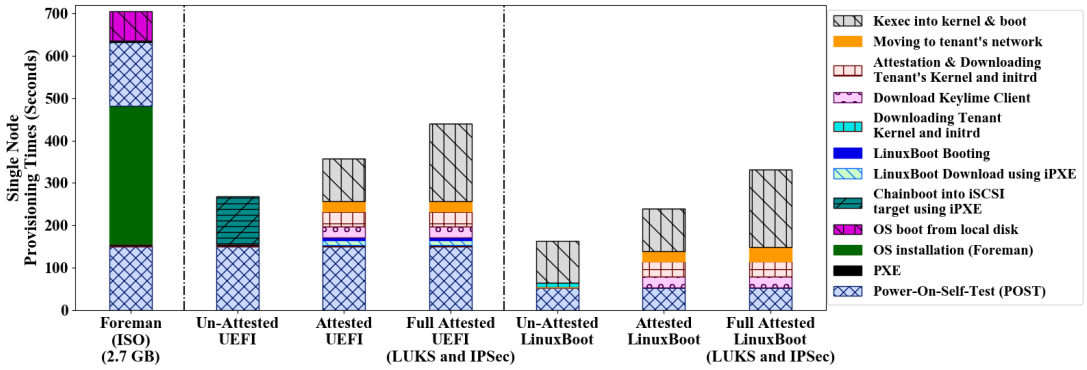


Figure A.4: Provisioning time of one server.

the system as they are used, most existing runtime protection measures like kernel integrity monitoring [264], control-flow integrity [265], or dynamic memory layout randomization [266] are built into either the kernel image/modules, application binaries, or libraries themselves. Thus, TPM measurements created by IMA at runtime will demonstrate that those protections were loaded.

After occupancy: Once a server is removed from a tenant enclave, we must ensure that the confidentiality of a tenant is not compromised by any of its state being visible to subsequent software running on the server. Stateless provisioning of the servers protects against any persistent state remaining on the server and avoids any reliance on the provider scrubbing the disk if it preempts the tenant. Further, the tenant can deploy its own provisioning service and ensure that the provider has no access to that storage. If the tenant requires the use of the local disks for performance reasons (e.g., for big data applications), the server can use local disk encryption with ephemeral keys stored only in memory. As long as the tenant attests that LinuxBoot is used, it knows that this firmware will zero the server's memory before another tenant will have the opportunity to execute any code.[7]

A.7 Evaluation

We first use micro-benchmarks to quantify the cost of encrypted storage and networking on our system, then examine the performance and scalability of the Bolted prototype, the cost of continuous attestation and finally the performance of applications deployed using Bolted under different assumptions of trust.

[7]Note that we are assuming here that the provider cannot re-flash the firmware remotely over the BMC.

A.7.1 Infrastructure and methodology

Single server provisioning experiments were performed on a Dell R630 server with 256 GB RAM and 2 Xeon E5-2660 v3 2.6GHz processors with 10 (20 HT) cores, using UEFI or LinuxBoot executing from motherboard flash. All the other experiments were conducted on a cluster of 16 Dell M620 blade servers (64 GB memory, 2 Xeon E5-2650 v2 2.60GHz processors with 8 cores (16 HT) per socket) and a 10Gbit switch. The M620 servers do not have a hardware TPM, so for functionality, we used a software emulation of a TPM [267], and for performance evaluation, emulated the latency to access the TPM based on numbers collected from our R630 system.

HIL, BMI, and Keylime servers were run on virtual machines with Xeon E5-2650 2.60GHz CPUs: Keylime with 16 vCPUs and 8GB memory; BMI with 2 vCPUs and 8GB, and HIL with 8 vCPUs and 8GB RAM. The iSCSI server ran on a virtual machine with 8 vCPUs and 32GB RAM. The Ceph cluster (the storage backend for BMI disk images) has 3 OSD servers (each dual Xeon E5-2603 v4 1.70GHz CPUs, 6 cores each) and a total of 27 disk spindles across the 3 machines. The servers were provisioned with Fedora 28 images (Linux kernel 4.17.9-200) enabled with IMA and version 5.6.3 of Strongswan [268] for IPsec. IPsec was configured in 'Host to Host' and Tunnel mode. The cryptographic algorithm used was AES-256-GCM SHA2-256 MODP2048. The authentication and encryption were done through a pre-shared key (PSK). IMA used SHA-256 hash algorithm. Cryptsetup utility version 1.7.0 was used to setup disk encryption based on LUKS – with AES-256-XTS algorithm. Unless otherwise stated, each experiment was executed five times.

A.7.2 The cost of encryption

For security-sensitive tenants that do not trust a provider, they must encrypt the disk and network traffic. To understand the overhead in our environment, we ran some simple micro-benchmarks.

Disk Encryption: Figure A.3a shows the overhead of LUKS disk encryption on a Block RAM disk exercised using Linux's "dd" command. While LUKS introduces overhead in this extreme case, we can see that the bandwidth that LUKS can sustain at 1GB for reads is likely to be able to keep up with both local disks and network mounted storage delivered over a 10Gbit network while write performance may introduce a modest degradation at ∼0.8GB.

Network Encryption: Figure A.3b shows the overhead of IPsec using Iperf between two servers using both hardware-based Intel AES-NI (IPsec HW) and software-based AES (IPsec SW) and MTU's of 1500 and 9000. We can see that IPsec has a much larger performance overhead than LUKS disk encryption, with even the best case of HW accelerated encryption and jumbo frames having almost a factor of two degradation over the non-encrypted case (CPU usage on our infrastructure is between 60% and 80% of one processing core for HW accelerated encryption). Additional tuning or specialized IPsec acceleration network interfaces could be used to boost performance [269]. We use hardware accelerated encryption and jumbo frames for all subsequent experiments.

Network mounted storage: In our implementation we boot servers using iSCSI which in turn accesses data from our Ceph cluster. In Figure A.3c we show the results of exercising the iSCSI server using "dd". Experimentally, we found that increasing the read ahead buffer size on Linux to 8MB was critical for performance, and we do this on all subsequent experiments (the default size is 128KB). Since Ceph as the backend storage reads data in 4MB chunks, increasing the read ahead buffer size to 8MB results in higher sequential read performance. As expected we find that LUKS introduces small overhead on writes and no overhead on reads, while IPsec between the client and iSCSI server has a major impact on performance.

A.7.3 Elasticity

Today's bare-metal clouds take many tens of minutes to allocate and provision a server [10]. Further, scrubbing the disk can take many hours; an operation required for stateful bare metal clouds whenever a server is being transferred between one tenant and another. In contrast, virtualized clouds are highly elastic; provisioning a new VM can take just a few minutes and deleting a VM is nearly instantaneous. The huge difference in elasticity between bare-metal clouds and virtualized clouds has a major impact on the use cases for which bare-metal clouds are appropriate. How close can we approach the elasticity of today's virtualized clouds? What extra cost does attestation impose on that elasticity? What is the extra cost if the tenant does not trust the provider and need to encrypt disks and storage?

To understand the elasticity Bolted supports, we first examine its performance for provisioning servers under different assumptions of security and then examine the concurrency

for provisioning multiple servers in parallel.

Provisioning time: Figure A.4 compares the time to provision a server with Foreman (a popular provisioning system) [270] to Bolted with both UEFI and LinuxBoot firmware under 3 scenarios: *no attestation* which would be used by clients that are insensitive to security, *attestation* where the tenant trusts the provider, but uses (provider deployed) attestation to ensure that previous tenants have not compromised the server, and *Full attestation*, where a security-sensitive tenant that does not trust the provider uses LUKS to encrypt the disk and IPsec to encrypt the path between the client and iSCSI server. There are a number of important high-level results from this figure. First for tenants that trust the provider, Bolted using LinuxBoot burned in the ROM is able to provision a server in under 3 minutes in the unattested case and under 4 minutes in the attested case; numbers that are very competitive with virtualized clouds. Second, attestation adds only a modest cost to provisioning a server and is likely a reasonable step for all systems. Third, even for tenants that do not trust the provider, (i.e. LUKS & IPsec) on servers with UEFI, Bolted at $\sim$7 minutes is still 1.6x faster than Foreman provisioning; note that Foreman implements no security procedures and is likely faster than existing cloud provisioning systems that use techniques like re-flashing firmware to protect tenants from firmware attacks.

Examining the detailed time breakdowns in Figure A.4; while we introduced LinuxBoot to improve security, we can see that the improved POST time (3x faster than UEFI) on these servers has a major impact on performance. We also see that booting from network mounted storage, introduced to avoid trusting the provider to scrub the disk, also has a huge impact on provisioning time. The time to install data on to the local disk is much larger for the Foreman case, where all data needs to be copied into the local disk. In contrast, with network booting, only a tiny fraction of the boot disk is ever accessed. We also see that with a stateful provisioning system like Foreman, it needs to reboot the server after installing the tenant's OS and applications on the local disk of the server; incurring POST time twice. While not explicitly shown here, it is also important to note that with Bolted a tenant can shutdown the OS and release a node to another tenant and then later restart the image on any compatible node; a key property of elasticity in virtualized clouds that is not possible with stateful provisioning systems like Foreman.

We show in Figure A.4 the costs of all the different phases of an attested boot. With UEFI, after POST, the phases are: (i) PXE downloading iPXE, (ii) iPXE downloading

LinuxBoot's runtime (Heads), (iii) booting LinuxBoot, (iv) downloading the Keylime Agent (using HTTP), (v) running the Keylime Agent, registering the server and attesting it, and then downloading the tenant's kernel and initrd, (vi) moving the server into the tenant's network and making sure it is connected to the BMI server and finally (vii) LinuxBoot kexec'ing into the tenant's kernel and booting the server. In each step, the running software measures the next software and extends the result into a TPM PCR. Using LinuxBoot firmware, after POST we immediately jump to step (iv) above.

While the steps for attestation where complex to implement, the overall performance cost is relatively modest, adding only around 25% to the cost of provisioning a server.[8] This is an important result given a large number of bare-metal systems (e.g. CloudLab, Chameleon, Foreman, ...), that take no security measure today to ensure that firmware has not been corrupted. There is no performance justification today for not using attestation, and our project has demonstrated that it is possible to measure all components needed to boot a server securely. For the full attestation scenarios (UEFI and LinuxBoot), two more steps are added to the basic attestation scenarios: (+i) loading the cryptographic key and decrypting the encrypted storage with LUKS (+ii) establishing IPsec tunnel and connecting to the encrypted network. These two steps are incorporated into Kernel boot time in Figure A.4. We can see that the major cost is not these extra steps but the slow down in booting into the image that comes from the slower disk that is accessed over IPsec.

Concurrency: Figure A.5 shows (with UEFI firmware) how Bolted performs, with and without attestation, as we increase the number of concurrently booting servers (log scale). In both the attested and unattested case performance stays relatively flat until 8 nodes. In our current environment, this level of concurrency/elasticity has been more than sufficient for the community of researchers using Bolted. There is a substantial degradation in both the attested and unattested case when we go from 8 to 16 servers. In the unattested case, the degradation is due to the small scale Ceph deployment (with only 27 disks) available in our experimental infrastructure. For the attested boot, the performance degradation arises from a limitation in our current implementation where we only support a single airlock at a time; attestation for provisioning is currently serialized. While this scalability limitation is not a problem for current use cases in our data center, we intend to address it to enable future

[8]Moreover, given that performance is sufficient, we have so far made no effort to optimize the implementation. Obvious opportunities include better download protocols than HTTP, porting the Keylime Agent from python to Rust, etc.

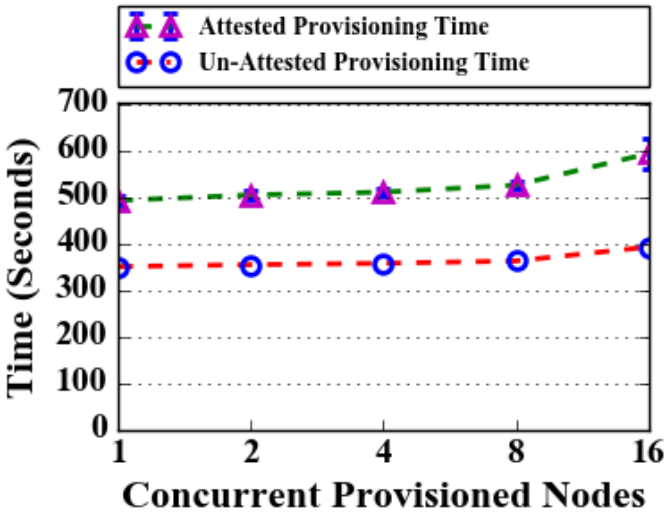


Figure A.5: Bolted Concurrency

use cases of highly-elastic security-sensitive tenants; e.g., a national emergency requiring many computers.

A.7.4 Continuous Attestation

Once a server has been provisioned, a security sensitive tenant can further use IMA to continuously measure any changes to the configuration and applications. The Keylime Agent will include the IMA measurement list along with periodic continuous attestation quotes. This allows the Keylime Cloud Verifier to help ensure the integrity of the server's runtime state by comparing the provided measurement list with a whitelist of approved values provided by the tenant. In the case of a policy violation, Keylime can then revoke any keys used for network or disk encryption; essentially isolating the server. To evaluate IMA performance, we measured Linux kernel 4.16.12 compile time with and without IMA with a different number of processing threads. We use kernel compilation as a test case for IMA because it requires extensive file I/O and execution of many binaries. The IMA policy we used measured all files that are executed as well as all files read by the *root* user. To stress IMA we ran the kernel compile as root such that all of its activity would be measured.[9] Figure A.6 shows the results in log scale; even in this unrealistic stress test IMA does not impose a noticeable overhead.

[9]This policy and workload are very unlikely to be either useful or manageable from a security perspective. We used them only as a stress test.

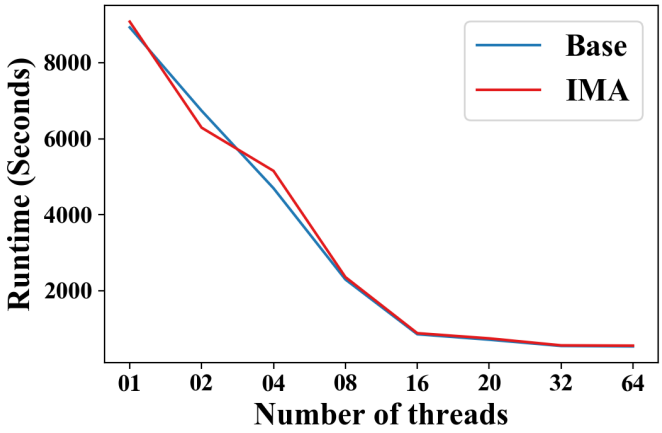


Figure A.6: IMA overhead on Linux Kernel Compile

Keylime can detect policy violations from checking the IMA measurements and TPM quotes in under one second. To simulate a policy violation, we ran a script on the server without having a record of it in the whitelist, resulting in an IMA measurement different than expected. This results in Keylime issuing a revocation notification for the key of the affected server used for IPsec to the other servers in the system; the entire process takes approximately 3 seconds for a compromised server to have its IPsec connections to other servers reset and be cryptographically banned from the network.

A.7.5 Macro-Benchmarks

Security-sensitive tenants using Bolted rely on network and disk encryption to minimize their trust in the provider. Surprisingly there is little information in the literature what the cost of such encryption is for real applications. Is the performance good enough that we can tolerate a one-size-fits-all solution and avoid ever trusting the provider? Is the performance so poor that it will never make sense for security-sensitive customers to use Bolted?

Figure A.7 (MPI) shows performance degradation results for a variety of applications from the NAS Parallel Benchmark [177] version 3.3.1: Embarrassingly Parallel (EP), Conjugate Gradient (CG), Fourier Transform (FT) and Multi Grid (MG) applications class D running in a 16 server enclave. We see overall that these applications only suffer significant overhead for IPsec, ranging from ∼18% for EB, which has modest communication, to ∼200% for CG which is very communication intensive. These results suggest that there are definitely workloads for which not trusting the provider incurs little overhead. At the

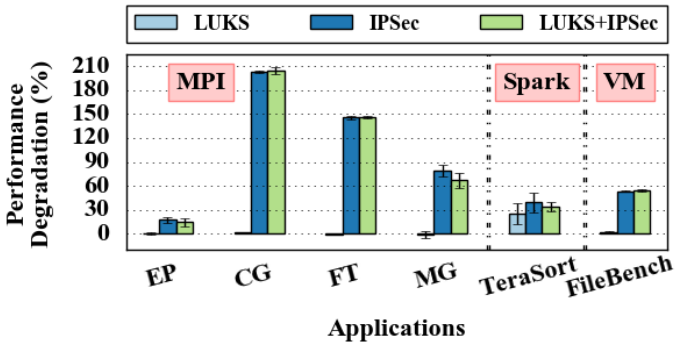


Figure A.7: Macro-benchmarks' performance

same time, a one-size-fits-all solution is inappropriate; only tenants that are willing to trust the provider, and avoid the cost of encryption, are likely to run highly communication intensive applications in the cloud.

To understand the performance overhead for more cloud relevant workloads, Figure A.7 (Spark) shows the performance of Spark [271] framework version 2.3.1 (working on Hadoop version 2.7.7) running TeraSort on a 260GB data set. The experiment is run in parallel in an enclave of 16 servers. TeraSort is a complex application which reads data from remote storage, shuffles temporary data between servers and writes final results to remote storage. We can see a significant overall degradation, of ∼30% for LUKS+IPsec. While this degradation is significant, we expect that security sensitive tenants would be willing to incur this level of overhead. On the other hand, this overhead is large enough that tenants willing to trust the provider would prefer not to incur it, suggesting that the flexibility of Bolted to provide this choice to the tenant is important.

Our last experiment (Figure A.7 (VM)) is based on virtualization. An important application of bare metal servers is to run virtualized software (e.g., an IaaS cloud). In this experiment, we installed KVM QEMU version 2.11.2 on a M620 server as the hypervisor. The virtual machine we run on the hypervisor is CentOS 7 with Linux kernel 3.10.0. It has 8 vCPU cores and 32 GB RAM. This is based on the observation [272] that 90% of virtual machines having $\leq$ 8 vCPU cores and $\leq$ 32 GB RAM. We run Filebench version 1.4.9.1 benchmark [273] on 1000 files with 12MB average size on the virtual machine. We can see that the performance of this benchmark is ∼50% worse in the case of IPsec; a significant performance penalty. While we would expect less of a degradation for regular VMs (rather than ones running a file system benchmark), we can see that a tenant deploying generic services, like virtualization, should be very careful about the kind of workload they expect

to use the service.

A.8 Related Work

Our work on creating a secure bare-metal cloud was motivated by a huge body of research demonstrating vulnerabilities due to co-location in virtualized clouds including both hypervisor attacks [219, 220, 221, 222, 223] and side-channel and cover-channel attacks like the Meltdown and Spectre exploits [198, 217, 218, 197].

There is a large body of products and research projects for bare-metal clouds [110, 111, 113, 225, 226] and cluster deployment systems [253, 254, 274, 275] that have many of the capabilities of isolation and provisioning that Bolted includes. The fundamental difference with Bolted, as we have explained in [276], is that we strongly separate isolation from provisioning and different entities (e.g. security sensitive tenants) can control/deploy and even re-implement the provisioning service. This structuring clearly defines the TCB that needs to be deployed by the provider.

While it is often unclear exactly which technique each cloud uses to protect against firmware attacks, a wide variety of techniques have been used including specialized hardware[277, 278], using a specialized hypervisor to prevent access to firmware [279], and attestation to the provider [280, 281]. In all cases, there is no way for a tenant to programmatically verify that the firmware is up to date and not compromised by previous tenants. Bolted is unique in enabling tenant deployed attestation for bare-metal servers, where the measurement of the firmware and software are provided directly to the tenant.

The static root of trust (SRTM) approach used by Bolted requires all software to be measured in an unbroken chain of trust. It would have been simpler for us to use dynamic root of trust (DRTM), however, DRTM has additional chip dependencies and, more importantly, been shown to be vulnerable to attacks [246] and work of Kovah et. al has shown that it can be used as an attack vector itself [282].

A.9 Discussion

We presented Bolted, an architecture for a bare metal cloud that is appropriate for even the most security sensitive tenants; allowing these customers to take control over their own security. The only trust these tenants need to place in the provider is for the availability

of the resources and that the physical hardware has not been compromised. At the same time, by delegating security for security sensitive tenants to the tenants, Bolted frees the provider from the complexity of having to directly support these demanding customers and avoids impact to customers that are less security sensitive.

To enable a wide community to inspect the TCB, all components of Bolted are open source. We designed HIL, for example, to be a simple micro-service rather than a general purpose tool like IRONIC [253] or Emulab [252]. HIL is being incorporated into a variety of different use cases by adding tools and services on and around it rather than turning it into a general purpose tool. Another key example of a small open source component is LinuxBoot. LinuxBoot is much simpler than UEFI. Since it is based on Linux, it has a code base that is under constant examination by a huge community of developers. LinuxBoot is reproducibly built, so a tenant can examine the software to ensure that it meets their security requirements and then ensure that the firmware deployed on machines is the version that they require.

Bolted protects against compromise of firmware executable by the system CPU; however modern systems may have other processors with persistent firmware inaccessible to the main CPU; compromise of this firmware is not addressed by this approach. These include: Base Management Controllers (BMCs) [283], the Intel Management Engine [284, 285, 286], PCIe devices with persistent flash-based firmware, like some GPUs and NICs, and storage devices [287]. Additional work (e.g. IOMMU based techniques, disabling the Management Engine [288] and the use of specialized systems with minimum firmware) will be needed to meet these threats.

The evaluation of our prototype has demonstrated that we can rapidly provision secure servers with competitive performance to today's virtualized clouds; removing one of the major barriers to bare metal clouds. We demonstrate that the cost of not trusting the provider (network/storage encryption) and of additional runtime security (continuous attestation) varies enormously depending on the application. (In fact, we are not aware of other work that has quantified the cost of network encryption, disk encryption, and continuous attestation with modern servers and implementation.) Results for HPC applications vary from negligible overhead to three times overhead for communication-intensive applications. Clearly the public cloud becomes economically unattractive for applications with three times overhead unless there are no other alternatives. However, we expect that the ∼30% degradation we see for TeraSort is likely representative of many applications today. Such

overheads suggest that the cost of security is modest enough that security-sensitive customers will find value in using cloud resources. At the same time, the overhead is significant enough that the flexibility of Bolted that enables tenants to just pay for the security they need is justified. One surprising result is that our secure firmware, LinuxBoot achieves dramatically better POST time than existing firmware; this is one of the few times in our experience that additional security comes with performance advantages.