Research Paper Reading Group

Pilot Session 1

Pilot overview

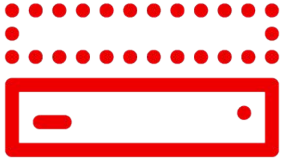
Session 1	4/3/2021	A perspective on research papers
Session 2	5/4/2021	Identifying worthwhile papers
Session 3	6/1/2021	Discussing research papers

Red Hat

What is a typical research paper

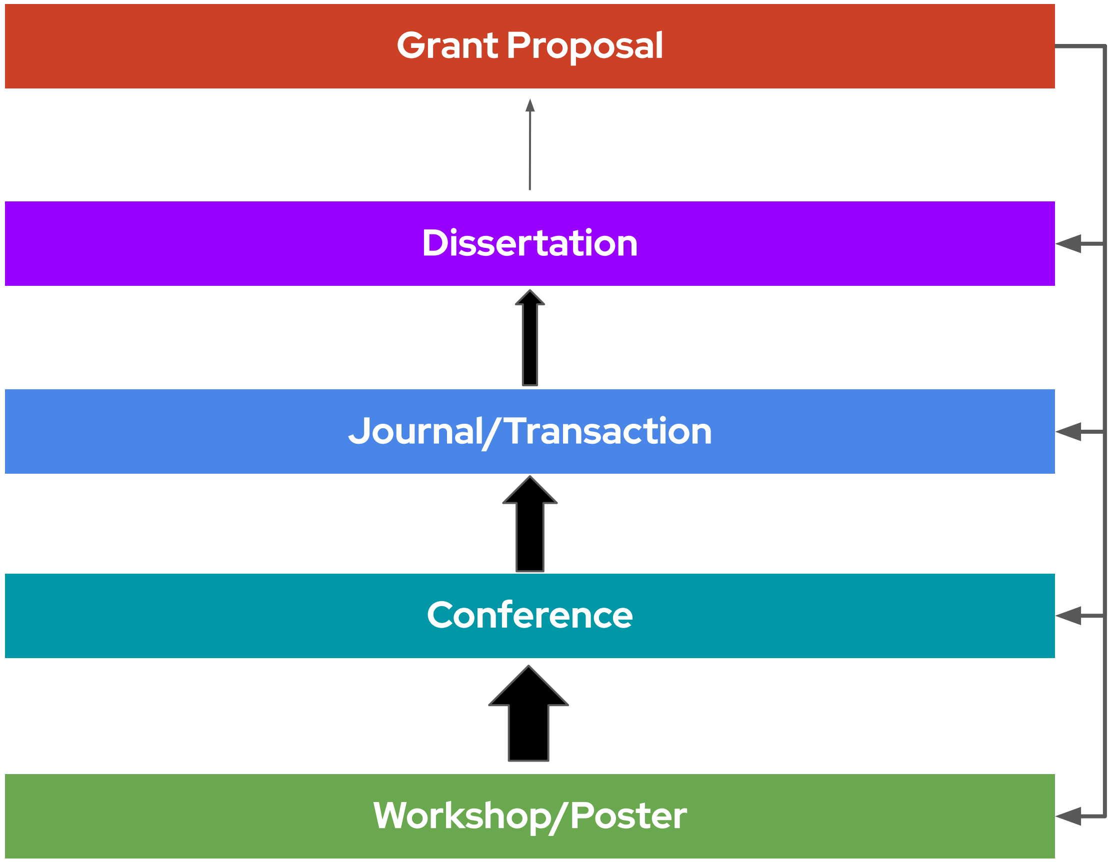
For computer science and engineering, it is a documentation of novel technical contributions in...

Applications/Services



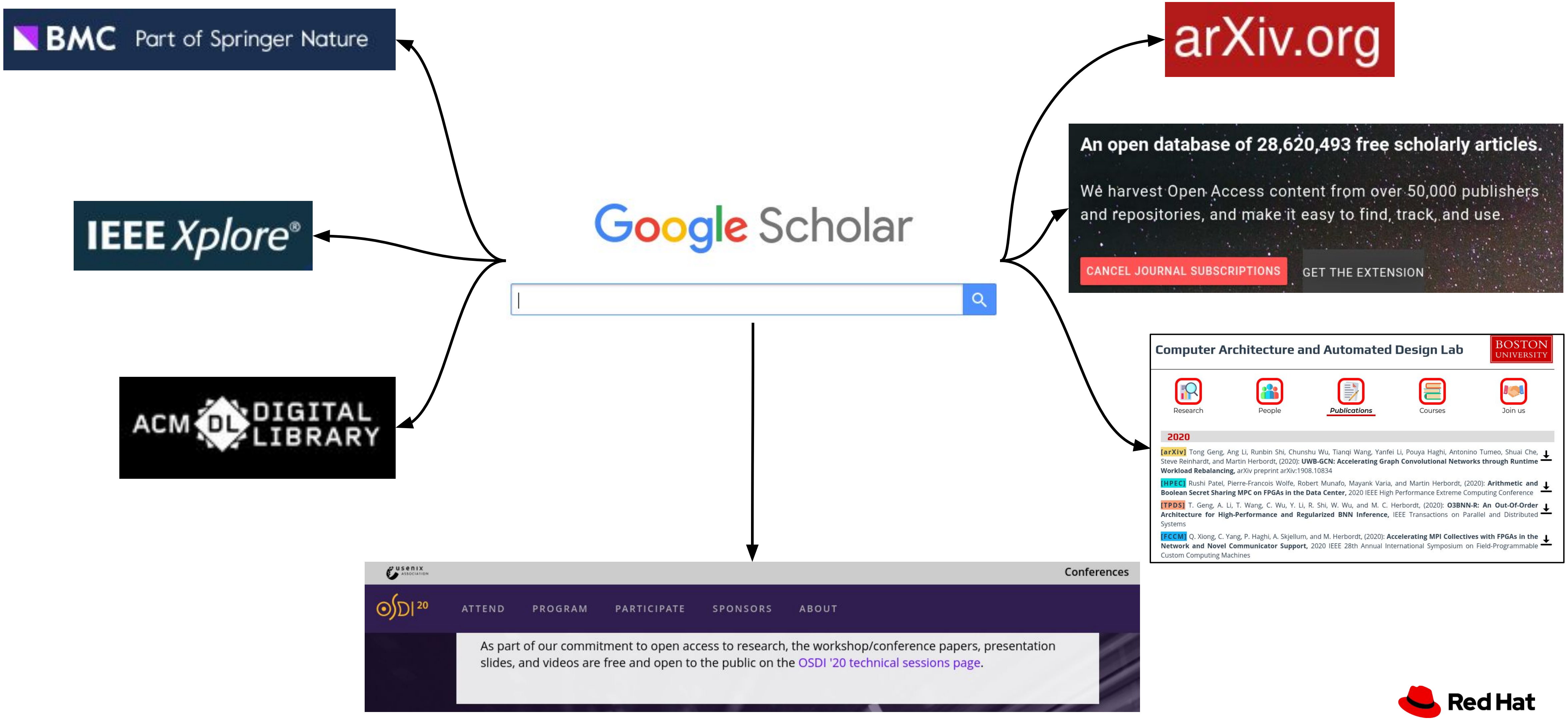
Systems

Types of research papers



Red Hat

Finding research papers



Red Hat

Tiers

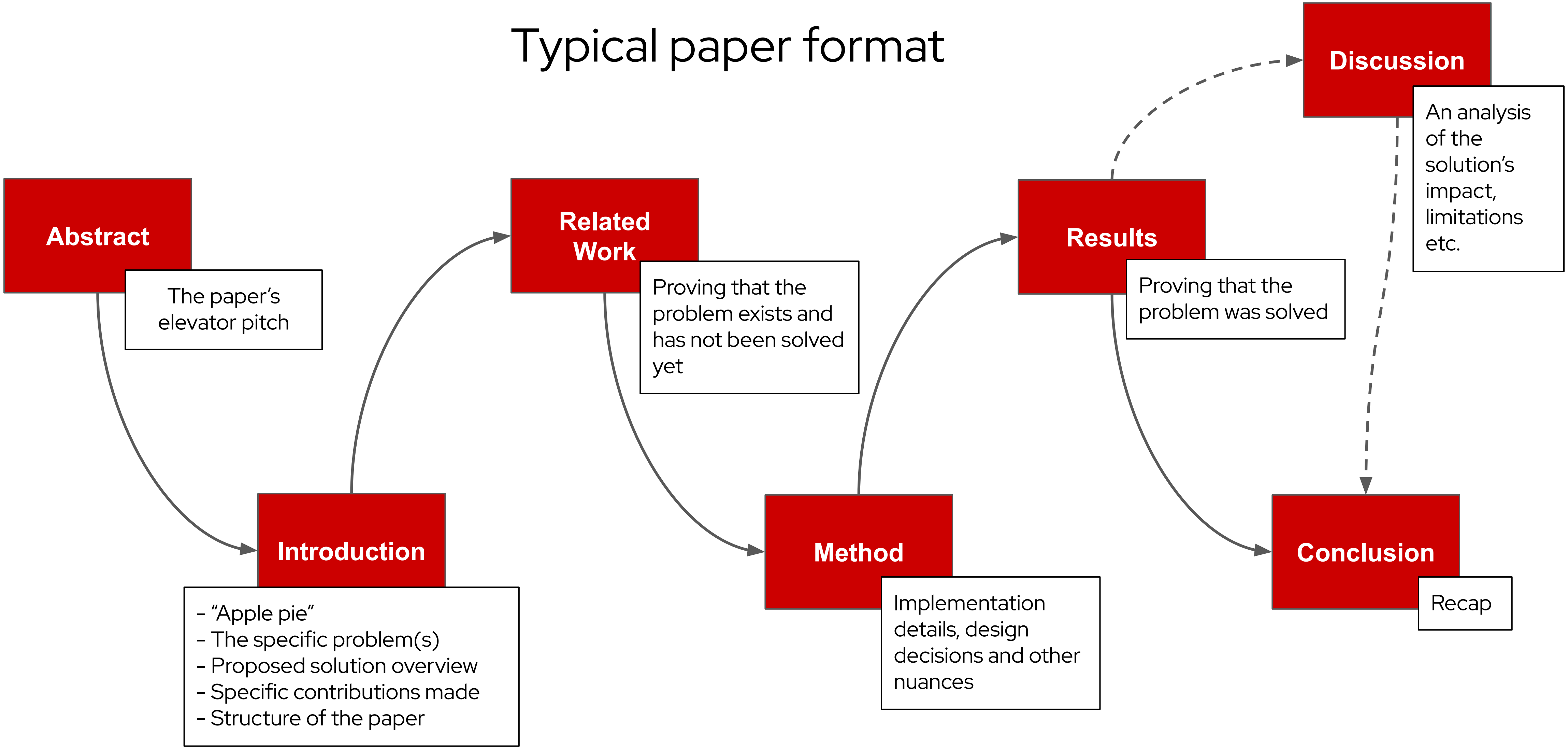
	Avg citations per paper	Conference
1.	66.3	CSUR—ACM Computing Surveys
2.	53.5	SOSP—ACM Symposium on Operating Systems Principles
3.	52.8	OSDI—Operating Systems Design and Implementation
4.	43.4	NDSS—Network and Distributed System Security Symposium
5.	40.9	MobiHoc—Mobile Ad Hoc Networking and Computing
6.	40.3	SIGCOMM—ACM SIGCOMM Conference
7.	38.2	SenSys—Conference On Embedded Networked Sensor Systems
8.	36.7	MOBICOM—Mobile Computing and Networking
9.	36.1	CIDR—Conference on Innovative Data Systems Research
10.	35.3	USENIX Security Symposium
11.	35.3	EUROCRYPT—Theory and Application of Cryptographic Techniques
12.	35.0	NSDI—Networked Systems Design and Implementation
13.	34.4	JASSS—The Journal of Artificial Societies and Social Simulation
14.	33.5	TOCS—ACM Transactions on Computer Systems
15.	33.5	S&P—IEEE Symposium on Security and Privacy
16.	33.4	MobiSys—International Conference on Mobile Systems
17.	32.5	IJCV—International Journal of Computer Vision
18.	32.2	TOG—ACM Transactions on Graphics/SIGGRAPH
19.	31.6	VLDB—Very Large Data Bases
20.	30.9	BioMED—Biomedical Engineering
21.	30.9	IEEE TRANS ROBOTICS AUTOMAT—IEEE Transactions on Robotics and Automation
22.	30.6	CRYPTO—International Crytology Conference
23.	30.1	PAMI—IEEE Transactions on Pattern Analysis and Machine Intelligence
24.	29.6	PLDI—SIGPLAN Conference on Programming Language Design and Implementation
25.	29.3	MICRO—International Symposium on Microarchitecture
26.	29.1	Journal of Web Semantics
27.	28.5	BIB—Briefings in Bioinformatics
28.	27.4	JMLR—Journal of Machine Learning Research
29.	27.0	ISMB—Intelligent Systems in Molecular Biology
30.	26.8	PODS—Symposium on Principles of Database Systems
31.	26.5	VTC—Vehicular Technology Conference
32.	25.5	SIGMOD—International Conference on Management of Data
33.	24.6	STOC—ACM Symposium on Theory of Computing
34.	24.0	TOIS—ACM Transactions on Information Systems
35.	24.0	IEEE SAP—IEEE Transactions on Speech and Audio Processing
36.	23.9	SCA—Symposium on Computer Animation
37.	23.3	BIOINFORMATICS—Bioinformatics/computer Applications in The Biosciences

[1] https://www.cs.cornell.edu/andru/csconf.html

The peer review process

[2] https://authorservices.wiley.com/asset/photos/Peer-Review-Process.pdf

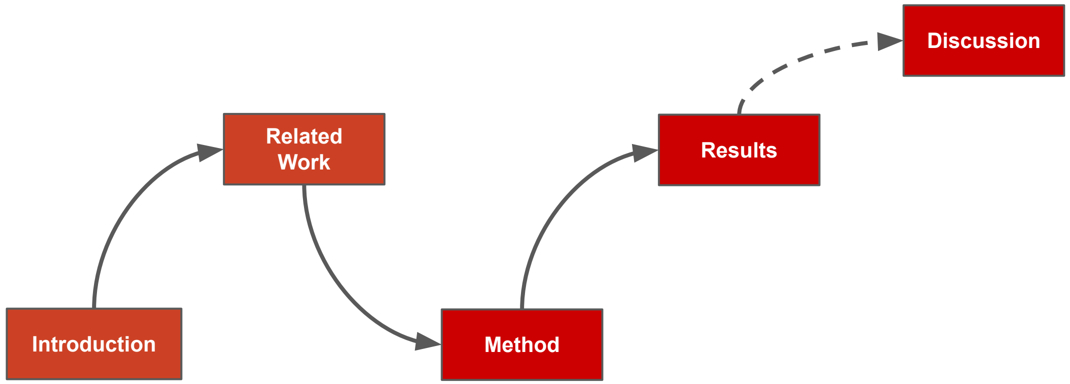
Typical paper format



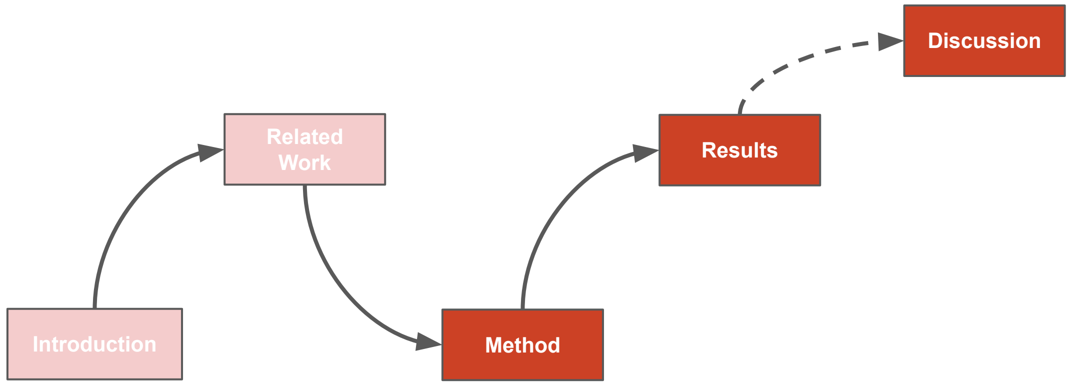
The value of research papers

The ‘specific’ value of research papers

Technical contributions

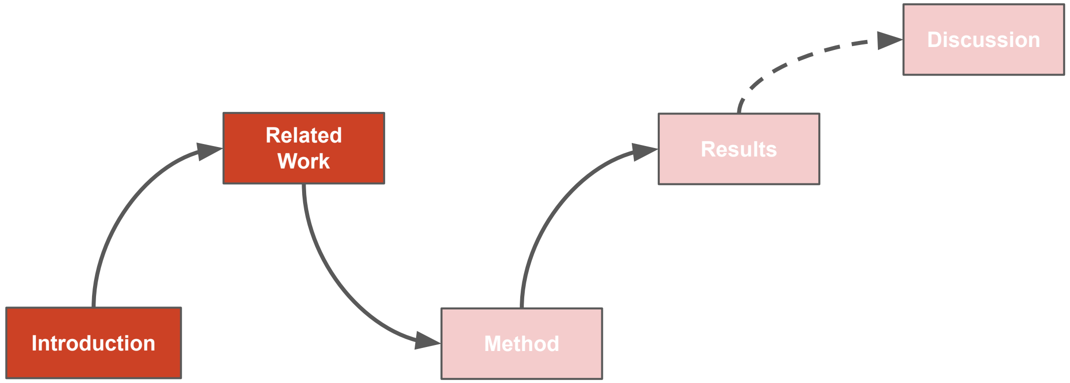


Potential future directions

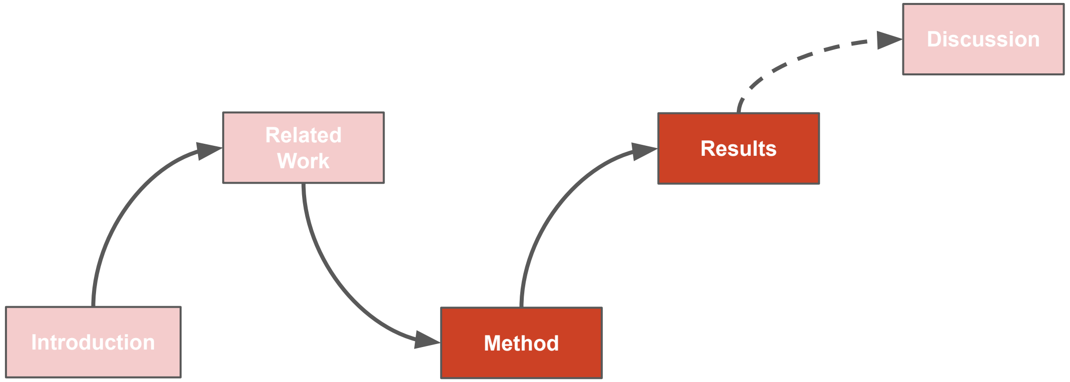


The ‘general’ value of research papers

Practical knowledge of the field



The scientific process



Case studies of general value: OSDI 2020

Paper 1: Scheduling track

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

Pradeep Ambati, *University of Massachusetts, Amherst;* Íñigo Goiri, Felipe Frujeri, *Microsoft Azure and Microsoft Research;* Alper Gun and Ke Wang, *Google;* Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini, *Microsoft Azure and Microsoft Research*

https://www.usenix.org/conference/osdi20/presentation/ambati

Paper 2: OS & Networking track

PANIC: A High-Performance Programmable NIC for Multi-tenant Networks

Jiaxin Lin, *University of Wisconsin - Madison;* Kiran Patel and Brent E. Stephens, *University of Illinois at Chicago;* Anirudh Sivaraman, *New York University (NYU);* Aditya Akella, *University of Wisconsin - Madison*

https://www.usenix.org/conference/osdi20/presentation/lin

Paper 3: Security track

Orchard: Differentially Private Analytics at Scale

Edo Roth, Hengchu Zhang, Andreas Haeberlen, and Benjamin C. Pierce, *University of Pennsylvania*

https://www.usenix.org/conference/osdi20/presentation/roth

Paper 1: Scheduling Track

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

Motivation. Cloud providers usually rent their resources to customers as Infrastructure as a Service (IaaS) VMs. When deployed, each VM consumes a fixed amount of resources from the server where it lands. Customers can keep their VMs for seconds or years [16] and may request more VMs over

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

Pradeep Ambati‡ Íñigo Goiri‡ Felipe Frujeri‡ Alper Gun†
Ke Wang† Brian Dolan† Brian Corell† Sekhar Pasupuleti†
Thomas Moscibroda† Sameh Elnikety‡ Marcus Fontoura† Ricardo Bianchini‡ *

†Microsoft Azure ‡Microsoft Research

Abstract

Cloud providers rent the resources they do not allocate as evictable virtual machines (VMs), like spot instances. In this paper, we first characterize the unallocated resources in Microsoft Azure, and show that they are plenty but may vary widely over time and across servers. Based on the characterization, we propose a new class of VM, called Harvest VM, to harvest and monetize the unallocated resources. A Harvest VM is more flexible and efficient than a spot instance, because it grows and shrinks according to the amount of unallocated resources at its underlying server; it is only evicted/killed when the provider needs its minimum set of resources. Next, we create models that predict the availability of the unallocated resources for Harvest VM deployments. Based on these predictions, we provide Service Level Objectives (SLOs) for the survival rate (*e.g.*, 65% of the Harvest VMs will survive more than a week) and the average number of cores that can be harvested. Our short-term predictions have an average error under 2% and less than 6% for longer terms. We also extend a popular cluster scheduling framework to leverage the harvested resources. Using our SLOs and framework, we can offset the rare evictions with extra harvested cores and achieve the same computational power as regular-priority VMs, but at 91% lower cost. Finally, we outline lessons and results from running Harvest VMs and our framework in production.

1 Introduction

Motivation. Cloud providers usually rent their resources to customers as Infrastructure as a Service (IaaS) VMs. When deployed, each VM consumes a fixed amount of resources from the server where it lands. Customers can keep their VMs for seconds or years [16] and may request more VMs over time. Thus, providers need to provide the illusion of perfectly elastic resources (*e.g.*, by reserving demand growth buffers) while operating the infrastructure with high availability (*e.g.*, by transparently handling hardware failures). For these reasons, they need to leave unallocated capacity.

To monetize this unallocated capacity, providers offer VMs with relaxed SLOs at discounted prices. Specifically, they offer low-priority evictable VMs, often called spot VMs [1, 8, 14]. These VMs are evicted if their resources are needed by regular-priority (or simply regular) on-demand VMs. Thus, evictable VMs are ideal for customers to run batch jobs or other workloads that can tolerate evictions, at very low cost.

Unfortunately, an evictable VM cannot consume all the unallocated resources of a server unless it fits perfectly in it. Even if it does, a large evictable VM will be promptly evicted whenever even a single resource is needed by a newly arriving regular VM. Multiple small evictable VMs can allocate the same amount of resources but will add overhead to operate more VMs. In addition, their larger number of evictions introduce VM re-creation and application re-initialization overheads that may even cause unavailability.

Given these limitations of existing evictable VMs, we argue that there should be a new class of evictable VMs able to *dynamically* and *flexibly* harvest all the unallocated resources of any server on which they land.

Our work. We first characterize the unallocated resources in Microsoft Azure. The characterization shows that there is potential for harvesting these resources, but they fluctuate over time and their availability is heterogeneous across servers and clusters. The characterization unearths the dynamics of the unallocated resources over multiple time durations.

Next, we propose a new class of evictable VM, called Harvest VM, as a novel way to monetize unallocated resources. A Harvest VM has a minimum size in terms of its physical resources, but it dynamically receives more or fewer physical resources beyond this minimum, depending on the amount of unallocated resources at its underlying server. A Harvest VM is only evicted if its minimum size is needed for a regular VM. In this paper, we focus on harvesting CPU cores.

Provisioning applications to run on harvested resources is challenging. However, we can predict the availability and amount of the unallocated resources in the datacenter. We use these predictions to provide SLOs for Harvest VM deployments. The SLO specifies the probability for a Harvest VM to survive for a certain period and how many resources it will get on average. For example, if a customer wants to create 100 VMs, the SLO may indicate that 90% of them

*Ambati is affiliated with the Univ. of Massachusetts Amherst, but was at Microsoft Research during this work. Gun and Wang are now with Google.

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

deployed, each VM consumes a fixed amount of resources from the server where it lands. Customers can keep their VMs for seconds or years [16] and may request more VMs over time. Thus, providers need to provide the illusion of perfectly

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

Pradeep Ambati[‡] Íñigo Goiri[‡] Felipe Frujeri[‡] Alper Gun[†]
Ke Wang[†] Brian Dolan[†] Brian Corell[†] Sekhar Pasupuleti[†]
Thomas Moscibroda[†] Sameh Elnikety[‡] Marcus Fontoura[†] Ricardo Bianchini[‡] *

[†]Microsoft Azure [‡]Microsoft Research

Abstract

Cloud providers rent the resources they do not allocate as evictable virtual machines (VMs), like spot instances. In this paper, we first characterize the unallocated resources in Microsoft Azure, and show that they are plenty but may vary widely over time and across servers. Based on the characterization, we propose a new class of VM, called Harvest VM, to harvest and monetize the unallocated resources. A Harvest VM is more flexible and efficient than a spot instance, because it grows and shrinks according to the amount of unallocated resources at its underlying server; it is only evicted/killed when the provider needs its minimum set of resources. Next, we create models that predict the availability of the unallocated resources for Harvest VM deployments. Based on these predictions, we provide Service Level Objectives (SLOs) for the survival rate (*e.g.*, 65% of the Harvest VMs will survive more than a week) and the average number of cores that can be harvested. Our short-term predictions have an average error under 2% and less than 6% for longer terms. We also extend a popular cluster scheduling framework to leverage the harvested resources. Using our SLOs and framework, we can offset the rare evictions with extra harvested cores and achieve the same computational power as regular-priority VMs, but at 91% lower cost. Finally, we outline lessons and results from running Harvest VMs and our framework in production.

1 Introduction

Motivation. Cloud providers usually rent their resources to customers as Infrastructure as a Service (IaaS) VMs. When deployed, each VM consumes a fixed amount of resources from the server where it lands. Customers can keep their VMs for seconds or years [16] and may request more VMs over time. Thus, providers need to provide the illusion of perfectly elastic resources (*e.g.*, by reserving demand growth buffers) while operating the infrastructure with high availability (*e.g.*, by transparently handling hardware failures). For these reasons, they need to leave unallocated capacity.

To monetize this unallocated capacity, providers offer VMs with relaxed SLOs at discounted prices. Specifically, they offer low-priority evictable VMs, often called spot VMs [1, 8, 14]. These VMs are evicted if their resources are needed by regular-priority (or simply regular) on-demand VMs. Thus, evictable VMs are ideal for customers to run batch jobs or other workloads that can tolerate evictions, at very low cost.

Unfortunately, an evictable VM cannot consume all the unallocated resources of a server unless it fits perfectly in it. Even if it does, a large evictable VM will be promptly evicted whenever even a single resource is needed by a newly arriving regular VM. Multiple small evictable VMs can allocate the same amount of resources but will add overhead to operate more VMs. In addition, their larger number of evictions introduce VM re-creation and application re-initialization overheads that may even cause unavailability.

Given these limitations of existing evictable VMs, we argue that there should be a new class of evictable VMs able to *dynamically* and *flexibly* harvest all the unallocated resources of any server on which they land.

Our work. We first characterize the unallocated resources in Microsoft Azure. The characterization shows that there is potential for harvesting these resources, but they fluctuate over time and their availability is heterogeneous across servers and clusters. The characterization unearths the dynamics of the unallocated resources over multiple time durations.

Next, we propose a new class of evictable VM, called Harvest VM, as a novel way to monetize unallocated resources. A Harvest VM has a minimum size in terms of its physical resources, but it dynamically receives more or fewer physical resources beyond this minimum, depending on the amount of unallocated resources at its underlying server. A Harvest VM is only evicted if its minimum size is needed for a regular VM. In this paper, we focus on harvesting CPU cores.

Provisioning applications to run on harvested resources is challenging. However, we can predict the availability and amount of the unallocated resources in the datacenter. We use these predictions to provide SLOs for Harvest VM deployments. The SLO specifies the probability for a Harvest VM to survive for a certain period and how many resources it will get on average. For example, if a customer wants to create 100 VMs, the SLO may indicate that 90% of them

*Ambati is affiliated with the Univ. of Massachusetts Amherst, but was at Microsoft Research during this work. Gun and Wang are now with Google.

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

> for seconds or years [16] and may request more VMs over time. Thus, providers need to provide the illusion of perfectly elastic resources (e.g., by reserving demand growth buffers) while operating the infrastructure with high availability (e.g., by transparently handling hardware failures). For these reasons, they need to leave unallocated capacity.

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

Pradeep Ambati[‡] Íñigo Goiri[‡] Felipe Frujeri[‡] Alper Gun[†]
Ke Wang[†] Brian Dolan[†] Brian Corell[†] Sekhar Pasupuleti[†]
Thomas Moscibroda[†] Sameh Elnikety[‡] Marcus Fontoura[†] Ricardo Bianchini[‡] [*]

[†]Microsoft Azure [‡]Microsoft Research

Abstract

Cloud providers rent the resources they do not allocate as evictable virtual machines (VMs), like spot instances. In this paper, we first characterize the unallocated resources in Microsoft Azure, and show that they are plenty but may vary widely over time and across servers. Based on the characterization, we propose a new class of VM, called Harvest VM, to harvest and monetize the unallocated resources. A Harvest VM is more flexible and efficient than a spot instance, because it grows and shrinks according to the amount of unallocated resources at its underlying server; it is only evicted/killed when the provider needs its minimum set of resources. Next, we create models that predict the availability of the unallocated resources for Harvest VM deployments. Based on these predictions, we provide Service Level Objectives (SLOs) for the survival rate (*e.g.*, 65% of the Harvest VMs will survive more than a week) and the average number of cores that can be harvested. Our short-term predictions have an average error under 2% and less than 6% for longer terms. We also extend a popular cluster scheduling framework to leverage the harvested resources. Using our SLOs and framework, we can offset the rare evictions with extra harvested cores and achieve the same computational power as regular-priority VMs, but at 91% lower cost. Finally, we outline lessons and results from running Harvest VMs and our framework in production.

1 Introduction

Motivation. Cloud providers usually rent their resources to customers as Infrastructure as a Service (IaaS) VMs. When deployed, each VM consumes a fixed amount of resources from the server where it lands. Customers can keep their VMs for seconds or years [16] and may request more VMs over time. Thus, providers need to provide the illusion of perfectly elastic resources (*e.g.*, by reserving demand growth buffers) while operating the infrastructure with high availability (*e.g.*, by transparently handling hardware failures). For these reasons, they need to leave unallocated capacity.

To monetize this unallocated capacity, providers offer VMs with relaxed SLOs at discounted prices. Specifically, they offer low-priority evictable VMs, often called spot VMs [1, 8, 14]. These VMs are evicted if their resources are needed by regular-priority (or simply regular) on-demand VMs. Thus, evictable VMs are ideal for customers to run batch jobs or other workloads that can tolerate evictions, at very low cost.

Unfortunately, an evictable VM cannot consume all the unallocated resources of a server unless it fits perfectly in it. Even if it does, a large evictable VM will be promptly evicted whenever even a single resource is needed by a newly arriving regular VM. Multiple small evictable VMs can allocate the same amount of resources but will add overhead to operate more VMs. In addition, their larger number of evictions introduce VM re-creation and application re-initialization overheads that may even cause unavailability.

Given these limitations of existing evictable VMs, we argue that there should be a new class of evictable VMs able to *dynamically* and *flexibly* harvest all the unallocated resources of any server on which they land.

Our work. We first characterize the unallocated resources in Microsoft Azure. The characterization shows that there is potential for harvesting these resources, but they fluctuate over time and their availability is heterogeneous across servers and clusters. The characterization unearths the dynamics of the unallocated resources over multiple time durations.

Next, we propose a new class of evictable VM, called Harvest VM, as a novel way to monetize unallocated resources. A Harvest VM has a minimum size in terms of its physical resources, but it dynamically receives more or fewer physical resources beyond this minimum, depending on the amount of unallocated resources at its underlying server. A Harvest VM is only evicted if its minimum size is needed for a regular VM. In this paper, we focus on harvesting CPU cores.

Provisioning applications to run on harvested resources is challenging. However, we can predict the availability and amount of the unallocated resources in the datacenter. We use these predictions to provide SLOs for Harvest VM deployments. The SLO specifies the probability for a Harvest VM to survive for a certain period and how many resources it will get on average. For example, if a customer wants to create 100 VMs, the SLO may indicate that 90% of them

*Ambati is affiliated with the Univ. of Massachusetts Amherst, but was at Microsoft Research during this work. Gun and Wang are now with Google.

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

while operating the infrastructure with high availability (*e.g.*, by transparently handling hardware failures). For these reasons, they need to leave unallocated capacity.

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

Pradeep Ambati[‡] Íñigo Goiri[‡] Felipe Frujeri[‡] Alper Gun[†]
Ke Wang[†] Brian Dolan[†] Brian Corell[†] Sekhar Pasupuleti[†]
Thomas Moscibroda[†] Sameh Elnikety[‡] Marcus Fontoura[†] Ricardo Bianchini[‡] [*]

[†]Microsoft Azure [‡]Microsoft Research

Abstract

Cloud providers rent the resources they do not allocate as evictable virtual machines (VMs), like spot instances. In this paper, we first characterize the unallocated resources in Microsoft Azure, and show that they are plenty but may vary widely over time and across servers. Based on the characterization, we propose a new class of VM, called Harvest VM, to harvest and monetize the unallocated resources. A Harvest VM is more flexible and efficient than a spot instance, because it grows and shrinks according to the amount of unallocated resources at its underlying server; it is only evicted/killed when the provider needs its minimum set of resources. Next, we create models that predict the availability of the unallocated resources for Harvest VM deployments. Based on these predictions, we provide Service Level Objectives (SLOs) for the survival rate (*e.g.*, 65% of the Harvest VMs will survive more than a week) and the average number of cores that can be harvested. Our short-term predictions have an average error under 2% and less than 6% for longer terms. We also extend a popular cluster scheduling framework to leverage the harvested resources. Using our SLOs and framework, we can offset the rare evictions with extra harvested cores and achieve the same computational power as regular-priority VMs, but at 91% lower cost. Finally, we outline lessons and results from running Harvest VMs and our framework in production.

1 Introduction

Motivation. Cloud providers usually rent their resources to customers as Infrastructure as a Service (IaaS) VMs. When deployed, each VM consumes a fixed amount of resources from the server where it lands. Customers can keep their VMs for seconds or years [16] and may request more VMs over time. Thus, providers need to provide the illusion of perfectly elastic resources (*e.g.*, by reserving demand growth buffers) while operating the infrastructure with high availability (*e.g.*, by transparently handling hardware failures). For these reasons, they need to leave unallocated capacity.

*Ambati is affiliated with the Univ. of Massachusetts Amherst, but was at Microsoft Research during this work. Gun and Wang are now with Google.

To monetize this unallocated capacity, providers offer VMs with relaxed SLOs at discounted prices. Specifically, they offer low-priority evictable VMs, often called spot VMs [1, 8, 14]. These VMs are evicted if their resources are needed by regular-priority (or simply regular) on-demand VMs. Thus, evictable VMs are ideal for customers to run batch jobs or other workloads that can tolerate evictions, at very low cost.

Unfortunately, an evictable VM cannot consume all the unallocated resources of a server unless it fits perfectly in it. Even if it does, a large evictable VM will be promptly evicted whenever even a single resource is needed by a newly arriving regular VM. Multiple small evictable VMs can allocate the same amount of resources but will add overhead to operate more VMs. In addition, their larger number of evictions introduce VM re-creation and application re-initialization overheads that may even cause unavailability.

Given these limitations of existing evictable VMs, we argue that there should be a new class of evictable VMs able to *dynamically* and *flexibly* harvest all the unallocated resources of any server on which they land.

Our work. We first characterize the unallocated resources in Microsoft Azure. The characterization shows that there is potential for harvesting these resources, but they fluctuate over time and their availability is heterogeneous across servers and clusters. The characterization unearths the dynamics of the unallocated resources over multiple time durations.

Next, we propose a new class of evictable VM, called Harvest VM, as a novel way to monetize unallocated resources. A Harvest VM has a minimum size in terms of its physical resources, but it dynamically receives more or fewer physical resources beyond this minimum, depending on the amount of unallocated resources at its underlying server. A Harvest VM is only evicted if its minimum size is needed for a regular VM. In this paper, we focus on harvesting CPU cores.

Provisioning applications to run on harvested resources is challenging. However, we can predict the availability and amount of the unallocated resources in the datacenter. We use these predictions to provide SLOs for Harvest VM deployments. The SLO specifies the probability for a Harvest VM to survive for a certain period and how many resources it will get on average. For example, if a customer wants to create 100 VMs, the SLO may indicate that 90% of them

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

To monetize this unallocated capacity, providers offer VMs with relaxed SLOs at discounted prices. Specifically, they offer low-priority evictable VMs, often called spot VMs [1, 8, 14]. These VMs are evicted if their resources are needed by regular-priority (or simply regular) on-demand VMs. Thus, evictable VMs are ideal for customers to run batch jobs or

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

Pradeep Ambati[‡] Íñigo Goiri[‡] Felipe Frujeri[‡] Alper Gun[†]
Ke Wang[†] Brian Dolan[†] Brian Corell[†] Sekhar Pasupuleti[†]
Thomas Moscibroda[†] Sameh Elnikety[‡] Marcus Fontoura[†] Ricardo Bianchini[‡] [*]

[†]Microsoft Azure [‡]Microsoft Research

Abstract

Cloud providers rent the resources they do not allocate as evictable virtual machines (VMs), like spot instances. In this paper, we first characterize the unallocated resources in Microsoft Azure, and show that they are plenty but m[illegible]ary widely over time and across servers. Based o[illegible] characterization, we propose a new class of V[illegible] called Harvest VM, to harvest and monetize th[illegible]located resources. A Harvest VM is more flexibl[illegible]efficient than a spot instance, because it grows a[illegible]hrinks according to the amount of unallocated r[illegible]rces at its underlying server; it is only evicted/killed when the provider needs its minimum set of resources. Next, we create models that predict the availability of the unallocated resources for Harvest VM deployments. Based on these predictions, we provide Service Level Objectives (SLOs) for the survival rate (*e.g.*, 65% of the Harvest VMs will survive more than a week) and the average number of cores that can be harvested. Our short-term predictions have an average error under 2% and less than 6% for longer terms. We also extend a popular cluster scheduling framework to leverage the harvested resources. Using our SLOs and framework, we can offset the rare evictions with extra harvested cores and achieve the same computational power as regular-priority VMs, but at 91% lower cost. Finally, we outline lessons and results from running Harvest VMs and our framework in production.

1 Introduction

Motivation. Cloud providers usually rent their resources to customers as Infrastructure as a Service (IaaS) VMs. When deployed, each VM consumes a fixed amount of resources from the server where it lands. Customers can keep their VMs for seconds or years [16] and may request more VMs over time. Thus, providers need to provide the illusion of perfectly elastic resources (*e.g.*, by reserving demand growth buffers) while operating the infrastructure with high availability (*e.g.*, by transparently handling hardware failures). For these reasons, they need to leave unallocated capacity.

To monetize this unallocated capacity, providers offer VMs with relaxed SLOs at discounted prices. Specifically, they offer low-priority evictable VMs, often called spot VMs [1, 8, 14]. These VMs are evicted if their resources are needed by regular-priority (or simply regular) on-demand VMs. Thus, evictable VMs are ideal for customers to run batch jobs or other workloads that can tolerate evictions, at very low cost.

Unfortunately, an evictable VM cannot consume all the unallocated resources of a server unless it fits perfectly in it. Even if it does, a large evictable VM will be promptly evicted whenever even a single resource is needed by a newly arriving regular VM. Multiple small evictable VMs can allocate the same amount of resources but will add overhead to operate more VMs. In addition, their larger number of evictions introduce VM re-creation and application re-initialization overheads that may even cause unavailability.

Given these limitations of existing evictable VMs, we argue that there should be a new class of evictable VMs able to *dynamically* and *flexibly* harvest all the unallocated resources of any server on which they land.

Our work. We first characterize the unallocated resources in Microsoft Azure. The characterization shows that there is potential for harvesting these resources, but they fluctuate over time and their availability is heterogeneous across servers and clusters. The characterization unearths the dynamics of the unallocated resources over multiple time durations.

Next, we propose a new class of evictable VM, called Harvest VM, as a novel way to monetize unallocated resources. A Harvest VM has a minimum size in terms of its physical resources, but it dynamically receives more or fewer physical resources beyond this minimum, depending on the amount of unallocated resources at its underlying server. A Harvest VM is only evicted if its minimum size is needed for a regular VM. In this paper, we focus on harvesting CPU cores.

Provisioning applications to run on harvested resources is challenging. However, we can predict the availability and amount of the unallocated resources in the datacenter. We use these predictions to provide SLOs for Harvest VM deployments. The SLO specifies the probability for a Harvest VM to survive for a certain period and how many resources it will get on average. For example, if a customer wants to create 100 VMs, the SLO may indicate that 90% of them

[*]Ambati is affiliated with the Univ. of Massachusetts Amherst, but was at Microsoft Research during this work. Gun and Wang are now with Google.

Red Hat

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

regular-priority (or simply regular) on-demand VMs. Thus, evictable VMs are ideal for customers to run batch jobs or other workloads that can tolerate evictions, at very low cost. Unfortunately, an evictable VM cannot consume all the

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

Pradeep Ambati[‡] Íñigo Goiri[‡] Felipe Frujeri[‡] Alper Gun[†]
Ke Wang[†] Brian Dolan[†] Brian Corell[†] Sekhar Pasupuleti[†]
Thomas Moscibroda[†] Sameh Elnikety[‡] Marcus Fontoura[†] Ricardo Bianchini[‡] *

[†]Microsoft Azure [‡]Microsoft Research

Abstract

Cloud providers rent the resources they do not allocate as evictable virtual machines (VMs), like spot instances. In this paper, we first characterize the unallocated resources in Microsoft Azure, and show that they are plenty but may vary widely over time and across servers. Based on the characterization, we propose a new class of VM, called Harvest VM, to harvest and monetize the unallocated resources. A Harvest VM is more flexible and efficient than a spot instance, because it grows and shrinks according to the amount of unallocated resources at its underlying server; it is only evicted/killed when the provider needs its minimum set of resources. Next, we create models that predict the availability of the unallocated resources for Harvest VM deployments. Based on these predictions, we provide Service Level Objectives (SLOs) for the survival rate (*e.g.*, 65% of the Harvest VMs will survive more than a week) and the average number of cores that can be harvested. Our short-term predictions have an average error under 2% and less than 6% for longer terms. We also extend a popular cluster scheduling framework to leverage the harvested resources. Using our SLOs and framework, we can offset the rare evictions with extra harvested cores and achieve the same computational power as regular-priority VMs, but at 91% lower cost. Finally, we outline lessons and results from running Harvest VMs and our framework in production.

1 Introduction

Motivation. Cloud providers usually rent their resources to customers as Infrastructure as a Service (IaaS) VMs. When deployed, each VM consumes a fixed amount of resources from the server where it lands. Customers can keep their VMs for seconds or years [16] and may request more VMs over time. Thus, providers need to provide the illusion of perfectly elastic resources (*e.g.*, by reserving demand growth buffers) while operating the infrastructure with high availability (*e.g.*, by transparently handling hardware failures). For these reasons, they need to leave unallocated capacity.

*Ambati is affiliated with the Univ. of Massachusetts Amherst, but was at Microsoft Research during this work. Gun and Wang are now with Google.

To monetize this unallocated capacity, providers offer VMs with relaxed SLOs at discounted prices. Specifically, they offer low-priority evictable VMs, often called spot VMs [1, 8, 14]. These VMs are evicted if their resources are needed by regular-priority (or simply regular) on-demand VMs. Thus, evictable VMs are ideal for customers to run batch jobs or other workloads that can tolerate evictions, at very low cost.

Unfortunately, an evictable VM cannot consume all the unallocated resources of a server unless it fits perfectly in it. Even if it does, a large evictable VM will be promptly evicted whenever even a single resource is needed by a newly arriving regular VM. Multiple small evictable VMs can allocate the same amount of resources but will add overhead to operate more VMs. In addition, their larger number of evictions introduce VM re-creation and application re-initialization overheads that may even cause unavailability.

Given these limitations of existing evictable VMs, we argue that there should be a new class of evictable VMs able to *dynamically* and *flexibly* harvest all the unallocated resources of any server on which they land.

Our work. We first characterize the unallocated resources in Microsoft Azure. The characterization shows that there is potential for harvesting these resources, but they fluctuate over time and their availability is heterogeneous across servers and clusters. The characterization unearths the dynamics of the unallocated resources over multiple time durations.

Next, we propose a new class of evictable VM, called Harvest VM, as a novel way to monetize unallocated resources. A Harvest VM has a minimum size in terms of its physical resources, but it dynamically receives more or fewer physical resources beyond this minimum, depending on the amount of unallocated resources at its underlying server. A Harvest VM is only evicted if its minimum size is needed for a regular VM. In this paper, we focus on harvesting CPU cores.

Provisioning applications to run on harvested resources is challenging. However, we can predict the availability and amount of the unallocated resources in the datacenter. We use these predictions to provide SLOs for Harvest VM deployments. The SLO specifies the probability for a Harvest VM to survive for a certain period and how many resources it will get on average. For example, if a customer wants to create 100 VMs, the SLO may indicate that 90% of them

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

...an evictable VM cannot consume all the unallocated resources of a server unless it fits perfectly in it.

...a large evictable VM will be promptly evicted whenever even a single resource is needed by a newly arriving regular VM.

Multiple small evictable VMs can allocate the same amount of resources but will add overhead to operate more VMs.

...larger number of evictions introduce VM re-creation and application re-initialization overheads that may even cause unavailability.

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

Pradeep Ambati[‡] Íñigo Goiri[‡] Felipe Frujeri[‡] Alper Gun[†]
Ke Wang[†] Brian Dolan[†] Brian Corell[†] Sekhar Pasupuleti[†]
Thomas Moscibroda[†] Sameh Elnikety[‡] Marcus Fontoura[†] Ricardo Bianchini[‡] *

[†]Microsoft Azure [‡]Microsoft Research

Abstract

Cloud providers rent the resources they do not allocate as evictable virtual machines (VMs), like spot instances. In this paper, we first characterize the unallocated resources in Microsoft Azure, and show that they are plenty but may vary widely over time and across servers. Based on the characterization, we propose a new class of VM, called Harvest VM, to harvest and monetize the unallocated resources. A Harvest VM is more flexible and efficient than a spot instance, because it grows and shrinks according to the amount of unallocated resources at its underlying server; it is only evicted/killed when the provider needs its minimum [illegible] resources. Next, we create [illegible] predict the availability of the unallocated resources for Harvest VM deployments. Based on these predictions, we provide Service Level Objectives (SLOs) for the survival rate (*e.g.*, 65% of the Harvest VMs will survive more than a week) and the average number of cores that can be harvested. Our short-term predictions have an average error under 2% and less than 6% for longer terms. We also extend a popular cluster scheduling framework to leverage the harvested resources. Using our SLOs and framework, we can offset the rare evictions with extra harvested cores and achieve the same computational power as regular-priority VMs, but at 91% lower cost. Finally, we outline lessons and results from running Harvest VMs and our framework in production.

1 Introduction

Motivation. Cloud providers usually rent their resources to customers as Infrastructure as a Service (IaaS) VMs. When deployed, each VM consumes a fixed amount of resources from the server where it lands. Customers can keep their VMs for seconds or years [16] and may request more VMs over time. Thus, providers need to provide the illusion of perfectly elastic resources (*e.g.*, by reserving demand growth buffers) while operating the infrastructure with high availability (*e.g.*, by transparently handling hardware failures). For these reasons, they need to leave unallocated capacity.

*Ambati is affiliated with the Univ. of Massachusetts Amherst, but was at Microsoft Research during this work. Gun and Wang are now with Google.

To monetize this unallocated capacity, providers offer VMs with relaxed SLOs at discounted prices. Specifically, they offer low-priority evictable VMs, often called spot VMs [1, 8, 14]. These VMs are evicted if their resources are needed by regular-priority (or simply regular) on-demand VMs. Thus, evictable VMs are ideal for customers to run batch jobs or other workloads that can tolerate evictions, at very low cost.

Unfortunately, an evictable VM cannot consume all the unallocated resources of a server unless it fits perfectly in it. Even if it does, a large evictable VM will be promptly evicted whenever even a single resource is needed by a newly arriving regular VM. Multiple small evictable VMs can allocate the same amount of resources but will add overhead to operate more VMs. In addition, their larger number of evictions introduce VM re-creation and application re-initialization overheads that may even cause unavailability.

Given these limitations of existing evictable VMs, we argue that there should be a new class of evictable VMs able to *dynamically* and *flexibly* harvest all the unallocated resources of any server on which they land.

Our work. We first characterize the unallocated resources in Microsoft Azure. The characterization shows that there is potential for harvesting these resources, but they fluctuate over time and their availability is heterogeneous across servers and clusters. The characterization unearths the dynamics of the unallocated resources over multiple time durations.

Next, we propose a new class of evictable VM, called Harvest VM, as a novel way to monetize unallocated resources. A Harvest VM has a minimum size in terms of its physical resources, but it dynamically receives more or fewer physical resources beyond this minimum, depending on the amount of unallocated resources at its underlying server. A Harvest VM is only evicted if its minimum size is needed for a regular VM. In this paper, we focus on harvesting CPU cores.

Provisioning applications to run on harvested resources is challenging. However, we can predict the availability and amount of the unallocated resources in the datacenter. We use these predictions to provide SLOs for Harvest VM deployments. The SLO specifies the probability for a Harvest VM to survive for a certain period and how many resources it will get on average. For example, if a customer wants to create 100 VMs, the SLO may indicate that 90% of them

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

> Given these limitations of existing evictable VMs, we argue that there should be a new class of evictable VMs able to *dynamically* and *flexibly* harvest all the unallocated resources of any server on which they land.

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

Pradeep Ambati‡ Íñigo Goiri‡ Felipe Frujeri‡ Alper Gun†
Ke Wang† Brian Dolan† Brian Corell† Sekhar Pasupuleti†
Thomas Moscibroda† Sameh Elnikety‡ Marcus Fontoura† Ricardo Bianchini‡ *

†Microsoft Azure ‡Microsoft Research

Abstract

Cloud providers rent the resources they do not allocate as evictable virtual machines (VMs), like spot instances. In this paper, we first characterize the unallocated resources in Microsoft Azure, and show that they are plenty but may vary widely over time and across servers. Based on the characterization, we propose a new class of VM, called Harvest VM, to harvest and monetize the unallocated resources. A Harvest VM is more flexible and efficient than a spot instance, because it grows and shrinks according to the amount of unallocated resources at its underlying server; it is only evicted/killed when the provider needs its minimum set of resources. Next, we create models that predict the availability of the unallocated resources for Harvest VM deployments. Based on these predictions, we provide Service Level Objectives (SLOs) for [illegible] 65% of the Harvest VMs will survive more than a week) and the average [illegible] that can be harvested. Our short-term predictions have an average error under 2% and less than 6% for longer terms. We also extend a popular cluster scheduling framework to leverage the harvested resources. Using our SLOs and framework, we can offset the rare evictions with extra harvested cores and achieve the same computational power as regular-priority VMs, but at 91% lower cost. Finally, we outline lessons and results from running Harvest VMs and our framework in production.

1 Introduction

Motivation. Cloud providers usually rent their resources to customers as Infrastructure as a Service (IaaS) VMs. When deployed, each VM consumes a fixed amount of resources from the server where it lands. Customers can keep their VMs for seconds or years [16] and may request more VMs over time. Thus, providers need to provide the illusion of perfectly elastic resources (*e.g.*, by reserving demand growth buffers) while operating the infrastructure with high availability (*e.g.*, by transparently handling hardware failures). For these reasons, they need to leave unallocated capacity.

To monetize this unallocated capacity, providers offer VMs with relaxed SLOs at discounted prices. Specifically, they offer low-priority evictable VMs, often called spot VMs [1, 8, 14]. These VMs are evicted if their resources are needed by regular-priority (or simply regular) on-demand VMs. Thus, evictable VMs are ideal for customers to run batch jobs or other workloads that can tolerate evictions, at very low cost.

Unfortunately, an evictable VM cannot consume all the unallocated resources of a server unless it fits perfectly in it. Even if it does, a large evictable VM will be promptly evicted whenever even a single resource is needed by a newly arriving regular VM. Multiple small evictable VMs can allocate the same amount of resources but will add overhead to operate more VMs. In addition, their larger number of evictions introduce VM re-creation and application re-initialization overheads that may even cause unavailability.

Given these limitations of existing evictable VMs, we argue that there should be a new class of evictable VMs able to *dynamically* and *flexibly* harvest all the unallocated resources of any server on which they land.

Our work. We first characterize the unallocated resources in Microsoft Azure. The characterization shows that there is potential for harvesting these resources, but they fluctuate over time and their availability is heterogeneous across servers and clusters. The characterization unearths the dynamics of the unallocated resources over multiple time durations.

Next, we propose a new class of evictable VM, called Harvest VM, as a novel way to monetize unallocated resources. A Harvest VM has a minimum size in terms of its physical resources, but it dynamically receives more or fewer physical resources beyond this minimum, depending on the amount of unallocated resources at its underlying server. A Harvest VM is only evicted if its minimum size is needed for a regular VM. In this paper, we focus on harvesting CPU cores.

Provisioning applications to run on harvested resources is challenging. However, we can predict the availability and amount of the unallocated resources in the datacenter. We use these predictions to provide SLOs for Harvest VM deployments. The SLO specifies the probability for a Harvest VM to survive for a certain period and how many resources it will get on average. For example, if a customer wants to create 100 VMs, the SLO may indicate that 90% of them

*Ambati is affiliated with the Univ. of Massachusetts Amherst, but was at Microsoft Research during this work. Gun and Wang are now with Google.

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

Background and related work

- VM deployments are partitioned into geographical regions and regions are partitioned into clusters of servers. Servers in a cluster have the same hardware but each region may have different number of clusters and hardware mix.
- There is a separate scheduler for region-level and cluster-level. Examples of scheduling factors: hardware required, maintenance tasks, available capacity.
- A server-level agent create the assigned VM and manages its lifecycle.
- Excess capacity is typically sold at discounted prices as evictabled VMs. Eviction notice varies between providers - AWS gives a 2 min warning, 30s for Azure.
- Dynamically changing virtual resources of a VM and enabling scheduler support for this can be unrealistic in practice. Simplicity and maintainability is important for production deployment.
- **Potential future work:** harvest any allocated cores that are temporarily idle.
- Traces of AWS EC2 spot prices are publically available and can be used to model the availability of spot instances - the challenge is the degree of accuracy and comprehensivity.
- It is better (and possible) to quantify unallocated resources at the granularity of a server, rather than aggregate data for the entire cluster.

will survive for more than 1 day, with an average of 10 cores. The provider does not monitor or actively seek to meet each individual SLO; instead, we retrain our prediction models frequently and provide our SLO as a statistical estimate [12]. As such, our SLOs can be considered predictions or estimates over large numbers of Harvest VMs, rather than guarantees.

Renting unallocated resources is cheap, but requires applications to manage the evictions. In addition, with Harvest VMs, the amount of resources backing each VM can vary. Harvest VMs are most useful when the applications they run can adapt to the number of available resources. For example, many applications use thread pools and can naturally adapt their parallelism. Others can schedule more load on larger VMs. The provider can hide these complexities by using Harvest VMs to create cheap SaaS (Software-as-a-Service), PaaS (Platform-as-a-Service), and FaaS (Function-as-a-Service) offerings. In fact, Harvest VMs are ideal for cluster scheduling (*e.g.*, Apache YARN [37], Kubernetes [22]) and serverless (*e.g.*, AWS Lambda [32], Azure Functions [4]) frameworks. These frameworks can schedule more tasks/functions on a Harvest VM that has grown to use more physical cores, and stop scheduling tasks/functions on one that has lost physical cores. To demonstrate how to adapt these frameworks, we build Harvest Hadoop to schedule computation (*e.g.*, data-processing, machine learning training) on harvested resources.

Our evaluation shows that we accurately predict the unallocated resources and provide SLOs. We predict the survival rate of a VM for 1 hour with an average error under 2% and lower than 6% for longer terms. We also predict the additional cores that can be harvested within a fraction of a core on average. Our SLOs and framework allow us to run Hadoop workloads on Harvest VMs at 91% lower cost to the customer than regular VMs, by offsetting the rare evictions with additional harvested cores. Compared to standard evictable VMs, the cost savings can reach 47%. Finally, we discuss lessons and results from deploying Harvest VMs and Harvest Hadoop in production to run internal workloads in Azure.

Summary. Our contributions are:

- We characterize the unallocated resources of a large cloud.
- We propose Harvest VMs to harvest unallocated resources.
- We build predictors for the availability of unallocated resources and provide a new SLO for these resources.
- We build Harvest Hadoop, a cluster scheduling framework to leverage Harvest VMs.
- We discuss lessons and results from our production deployment of Harvest VMs and Harvest Hadoop.

2 Background and related work

Deploying VMs. Each VM deployment targets a geographical region, which is partitioned into clusters of servers that have the same hardware. Each region may have a different number of clusters and hardware mix. A region-level scheduler decides which VMs go to which clusters based on several factors (*e.g.*, hardware required, maintenance tasks, available capacity) [19]. These factors can cause clusters to have different VM loads, even in the same region. Then, a cluster-level scheduler decides which server in the cluster will run each VM. When a VM is assigned to a server, a server-level agent creates the VM and manages its lifecycle.

Evictable VMs. Providers sell their excess capacity at discounted prices as evictable VMs [1, 8, 14]. These VMs are evicted/killed when the provider needs the capacity (*e.g.*, due to a spike in the number of on-demand VMs). Providers notify the VMs before they evict them: GCP and Azure provide a 30-second warning, whereas AWS gives 2 minutes.

Variable-resource VMs. Sharma *et al.* [33] recently proposed Deflatable VMs, which change *virtual* resources dynamically (via hot-plugging/unplugging), and a multi-level resource reclamation approach for explicitly adapting applications, operating systems, and hypervisors to the available resources. They also combined reclamation with deflation-aware VM scheduling. We believe that expecting the whole stack to adapt is unrealistic in practice. Instead, we favor simplicity and maintainability for *production deployment*: (1) we minimize the changes to the cloud platform, so deploying Harvest VMs is no different than deploying any other VM, and the VM scheduler is unaware that Harvest VMs grow and shrink; (2) we do not change the number of virtual cores, and instead transparently vary the number of physical cores.

A more aggressive VM design could harvest the unallocated cores *and* any allocated cores that are temporarily idle. This is out of the scope of this paper. Instead, we focus on the usability of core-harvesting VMs (aggressive or otherwise) in practice with SLOs and software for them. Our SLOs can be extended for aggressive harvesting, whereas Harvest Hadoop can be used directly.

Like a Harvest VM, a burstable VM [7, 13] has a fixed number of virtual cores and receives a minimum number of physical cores. However, it is only allowed to burst (*i.e.*, receive additional physical cores) up to its maximum size, after accumulating enough "credits" by staying below a predefined core utilization. A Harvest VM differs in that (1) it harvests as many cores as are unallocated for as long as they remain so, *i.e.* there is no concept of credit; and (2) it is evictable. These characteristics mean that providing SLOs for Harvest VMs is also quite different than for burstable VMs.

Resource harvesting. Other approaches to resource harvesting have either focused on running batch workloads on idle machines (*e.g.* [25, 26]) or co-locating batch workloads with latency-sensitive services on bare-metal servers (*e.g.* [23, 27, 38, 39, 46, 47]). In contrast, we focus on a virtualized infrastructure where physical resources are reserved for the VMs that allocate them (as is the norm in the public cloud), and predict the availability and dynamics of the unallocated resources to produce SLOs.

Characterization and SLOs. To indirectly characterize the unallocated resources at cloud providers, prior work [2, 9, 31,

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

Characterizing unallocated resources: Azure 2/19 - 10/19 data	Methodology, Temporal patterns, Cluster behaviors, Regional aggregated data, Minimum unallocated cores, Additional unallocated cores, Multiple VMs per server, High-level takeaways
Proposed VM class: Harvest VM	Overview, Production implementation approach, Comparison to standard evictable VMs, Workload/application requirements, Privacy/Confidentiality, Pricing, Harvesting resources other than cores
Prediction for survival rate: ML based approach	User input, ML models and features, ML training and inference, Discarded features, Applying prediction to standard evictable VM survivability
Scheduler support: Harvest Hadoop	Architecture, Eviction management, Core reassignment management, Harvesting resources other than cores
Evaluation	Evaluation focus, Simulator, Experiments, Analysis (benefits, accuracy, scheduler, cost)

Paper 2: OS & Networking Track

PANIC: A High-Performance Programmable NICfor Multi-tenant Networks

Red Hat

PANIC: A High-Performance Programmable NICfor Multi-tenant Networks

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

PANIC: A High-Performance Programmable NIC for Multi-tenant Networks

Jiaxin Lin
University of Wisconsin-Madison

Kiran Patel
University of Illinois at Chicago

Brent E. Stephens
University of Illinois at Chicago

Anirudh Sivaraman
New York University (NYU)

Aditya Akella
University of Wisconsin-Madison

A Artifact Appendix

A.1 Abstract

This artifact contains the source code and test benches for PANIC's 100Gbps FPGA-based prototype. Our FPGA prototype is implemented in pure Verilog. Features of the prototype include: the hybrid push/pull packet scheduler, the high-performance switching interconnect, self-contained compute units, and the lightweight RMT pipeline.

This artifact provides two test benches to reproduce the results in Figure 8c and Figure 11a in the Vivado HDL simulator.

A.2 Artifact check-list

- **Compilation:** Running this artifact requires Vivado Design Suite [10]. Vivado v2019.x and v2020.1 WebPack are verified.
- **Hardware:** This artifact does not requires any specific hardware.
- **Metrics:** This artifact measures PANIC's receiving throughput under different chaining models and traffic patterns.
- **Output:** The result will be printed to the console and log files.
- **Experiments:** This artifact includes testbenches and running scripts to replay Figure 8c and Figure 11a.
- **Public link:** https://bitbucket.org/uw-madison-networking-research/panic_osdi20_artifact

A.3 Description

A.3.1 How to access

This artifact is publicly available at https://bitbucket.org/uw-madison-networking-research/panic_osdi20_artifact.

A.3.2 Software dependencies

Running this artifact requires Vivado [10]. Vivado WebPack version is license-free, and it has simulation capabilities to recreate our results. Since installing the Vivado WebPack requires plenty of disk space (>20GB), you can choose to instance an FPGA Developer AMI in AWS (https://aws.amazon.com/marketplace/pp/B06VVYBLZZ) to run this artifact. The FPGA Developer AMI has pre-installed the required Vivado toolchain.

A.4 Experiment workflow

1. **Check Vivado is Installed Correctly**

```
$ vivado -mode tcl
// Enter the Vivado Command Palette
Vivado% version
// v2019.x and v2020.1 is verified
Vivado% quit
```

2. **Clone the Repo and Make Run**

```
$ git clone [Artifact_Repo]
$ cd panic_osdi20_artifact
$ make test_parallel
$ make test_shaaes
```

The make command first compiles the source code, then runs the simulation tasks in Vivado. The *test_parallel* test replays Figure 8c and the *test_shaaes* test replays Figure 11a.

A.5 Evaluation and expected result

The result will be printed to the console. The output will also be logged in *./build/export_sim/xsim/simulate.log*. For the expected output and analysis please reference Figure 8c and Figure 11a.

A.6 Notes

For more details about the code structure, please reference https://bitbucket.org/uw-madison-networking-research/panic_osdi20_artifact/src/master/README.md

A.7 AE Methodology

Submission, reviewing and badging methodology:

- https://www.usenix.org/conference/osdi20/call-for-artifacts

Red Hat

PANIC: A High-Performance Programmable NICfor Multi-tenant Networks

“Apple pie”

- The gap between network line-rates and the rate at which a CPU can produce and consume data is widening rapidly.
- Emerging programmable (“smart”) NICs can help overcome this problem.
- There are many different types of offloads that can be implemented on a programmable NIC.
- These offloads, which accelerate computation across all of the different layers of the network stack, can reduce load on the general purpose CPU, reduce latency, and increase throughput
- Many different cloud and datacenter applications and use cases have been shown to benefit from offloading computation to programmable NICs

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE | ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL | ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

PANIC: A High-Performance Programmable NIC for Multi-tenant Networks

Jiaxin Lin
University of Wisconsin-Madison

Kiran Patel
University of Illinois at Chicago

Brent E. Stephens
University of Illinois at Chicago

Anirudh Sivaraman
New York University (NYU)

Aditya Akella
University of Wisconsin-Madison

Abstract

Programmable NICs have diverse uses, and there is a need for a NIC platform that can offload computation from multiple co-resident applications to many different types of substrates, including hardware accelerators, embedded FPGAs, and embedded processor cores. Unfortunately, there is no existing NIC design that can simultaneously support a large number of diverse offloads while ensuring high throughput/low latency, multi-tenant isolation, flexible offload chaining, and support for offloads with variable performance.

This paper presents PANIC, a new programmable NIC. There are two new key components of the PANIC design that enable it to overcome the limitations of existing NICs: 1) A high-performance switching interconnect that scalably connects independent engines into offload chains, and 2) A new hybrid push/pull packet scheduler that provides cross-tenant performance isolation and low-latency load-balancing across parallel offload engines. From experiments performed on an 100 Gbps FPGA-based prototype, we find that this design overcomes the limitations of state-of-the-art programmable NICs.

1 Introduction

The gap between network line-rates and the rate at which a CPU can produce and consume data is widening rapidly [71, 66]. Emerging *programmable (“smart”) NICs* can help overcome this problem [32]. There are many different types of offloads that can be implemented on a programmable NIC. These offloads, which accelerate computation across all of the different layers of the network stack, can reduce load on the general purpose CPU, reduce latency, and increase throughput [32, 48, 59, 69, 13].

Many different cloud and datacenter applications and use cases have been shown to benefit from offloading computation to programmable NICs [13, 48, 59, 42, 32, 37, 49, 46, 62, 47, 30, 36, 70, 69, 35, 55, 45]. However, there is no single “silver bullet” offload that can improve performance in all cases. Instead, we anticipate that different applications will specify their own chains of offloads, and that the operator will then merge these chains with infrastructure-related offloads and run them on her programmable NICs. To realize this vision, this paper presents PANIC, a new scalable and high-performance programmable NIC for multi-tenant networks that supports a wide variety of different types of offloads and composes them into isolated offload chains.

To enable cloud operators to provide NIC offload chains as a service to tenants, a programmable NIC must support: 1) Offload variety: some offloads like cryptography are best suited for hardware implementations, while an offload providing a low-latency bypass for RPCs in an application is better suited for an embedded core [51]; 2) Offload chaining: to minimize wasted chip area on redundant functions, the NIC should facilitate composing independent hardware offload units into a chain as needed, with commonly-needed offloads shared across tenants; 3) Multi-tenant isolation: tenants should not be able to consume more than their allocation of a shared offload; 4) Variable-performance offloads: there are useful offloads that are not guaranteed to run at line-rate, as well as important offloads that run with low latency and at line-rate.

There exist many different programmable NICs [32, 12, 75, 31, 58, 72, 23, 24, 11, 57, 53, 54, 52, 76], but, there is no programmable NIC that is currently able to provide all of the above properties. Existing NIC designs can be categorized as follows, with each category imposing key limitations:

- **Pipeline-of-Offloads NICs** place multiple offloads in a pipeline to enable packets to be processed by a chain of functions [52, 32]. Chaining can be modified in these NICs today but requires a significant amount of time and developer effort for FPGA synthesis, and slow offloads cause packet loss or head-of-line (HOL) blocking.
- **Manycore NICs** load balance packets across many embedded CPU cores, with the CPU core then controlling the processing of packets as needed for different offloads [23, 24, 53, 54, 57, 72, 58]. These designs suffer from performance issues because embedded CPU cores add tens of microseconds of additional latency [32]. Also, no existing manycore NICs provide performant mechanisms to isolate competing tenants. Further, performance on manycore NICs can degrade significantly if the working set does not fit within the core's cache.
- **RMT NICs** use on-NIC reconfigurable match+action (RMT) pipeline to implement NIC offloads. The types of offloads that can be supported by RMT pipelines are limited because each pipeline stage must be able to handle processing a new packet every single clock cycle.

PANIC: A High-Performance Programmable NICfor Multi-tenant Networks

Problems/Challenges

- No existing programmable NIC that supports all of the following properties:
  - Offload variety
    - some offloads like cryptography are best suited for hardware implementations, while an offload providing a low-latency bypass for RPCs in an application is better suited for an embedded core
  - Offload chaining
    - to minimize wasted chip area on redundant functions, the NIC should facilitate composing independent hardware offload units into a chain as needed, with commonly-needed offloads shared across tenants
  - Multi-tenant isolation
    - tenants should not be able to consume more than their allocation of a shared offload
  - Variable-performance offloads
    - there are useful offloads that are not guaranteed to run at line-rate, as well as important offloads that run with low latency and at line-rate.


ARTIFACT EVALUATED usenix AVAILABLE | ARTIFACT EVALUATED usenix FUNCTIONAL | ARTIFACT EVALUATED usenix REPRODUCED

PANIC: A High-Performance Programmable NIC for Multi-tenant Networks

Jiaxin Lin, *University of Wisconsin-Madison*
Kiran Patel, *University of Illinois at Chicago*
Brent E. Stephens, *University of Illinois at Chicago*
Anirudh Sivaraman, *New York University (NYU)*
Aditya Akella, *University of Wisconsin-Madison*

Abstract

Programmable NICs have diverse uses, and there is a need for a NIC platform that can offload computation from multiple co-resident applications to many different types of substrates, including hardware accelerators, embedded FPGAs, and embedded processor cores. Unfortunately, there is no existing NIC design that can simultaneously support a large number of diverse offloads while ensuring high throughput/low latency, multi-tenant isolation, flexible offload chaining, and support for offloads with variable performance.

This paper presents PANIC, a new programmable NIC. There are two new key components of the PANIC design that enable it to overcome the limitations of existing NICs: 1) A high-performance switching interconnect that scalably connects independent engines into offload chains, and 2) A new hybrid push/pull packet scheduler that provides cross-tenant performance isolation and low-latency load-balancing across parallel offload engines. From experiments performed on an 100 Gbps FPGA-based prototype, we find that this design overcomes the limitations of state-of-the-art programmable NICs.

1 Introduction

The gap between network line-rates and the rate at which a CPU can produce and consume data is widening rapidly [71, 66]. Emerging *programmable ("smart") NICs* can help overcome this problem [32]. There are many different types of offloads that can be implemented on a programmable NIC. These offloads, which accelerate computation across all of the different layers of the network stack, can reduce load on the general purpose CPU, reduce latency, and increase throughput [32, 48, 59, 69, 13].

Many different cloud and datacenter applications and use cases have been shown to benefit from offloading computation to programmable NICs [13, 48, 59, 42, 32, 37, 49, 46, 62, 47, 30, 36, 70, 69, 35, 55, 45]. However, there is no single "silver bullet" offload that can improve performance in all cases. Instead, we anticipate that different applications will specify their own chains of offloads, and that the operator will then merge these chains with infrastructure-related offloads and run them on her programmable NICs. To realize this vision, this paper presents PANIC, a new scalable and high-performance programmable NIC for multi-tenant networks that supports a wide variety of different types of offloads and composes them into isolated offload chains.

To enable cloud operators to provide NIC offload chains as a service to tenants, a programmable NIC must support: 1) Offload variety: some offloads like cryptography are best suited for hardware implementations, while an offload providing a low-latency bypass for RPCs in an application is better suited for an embedded core [51]; 2) Offload chaining: to minimize wasted chip area on redundant functions, the NIC should facilitate composing independent hardware offload units into a chain as needed, with commonly-needed offloads shared across tenants; 3) Multi-tenant isolation: tenants should not be able to consume more than their allocation of a shared offload; 4) Variable-performance offloads: there are useful offloads that are not guaranteed to run at line-rate, as well as important offloads that run with low latency and at line-rate.

There exist many different programmable NICs [32, 12, 75, 31, 58, 72, 23, 24, 11, 57, 53, 54, 52, 76], but, there is no programmable NIC that is currently able to provide all of the above properties. Existing NIC designs can be categorized as follows, with each category imposing key limitations:

- **Pipeline-of-Offloads NICs** place multiple offloads in a pipeline to enable packets to be processed by a chain of functions [52, 32]. Chaining can be modified in these NICs today but requires a significant amount of time and developer effort for FPGA synthesis, and slow offloads cause packet loss or head-of-line (HOL) blocking.
- **Manycore NICs** load balance packets across many embedded CPU cores, with the CPU core then controlling the processing of packets as needed for different offloads [23, 24, 53, 54, 57, 72, 58]. These designs suffer from performance issues because embedded CPU cores add tens of microseconds of additional latency [32]. Also, no existing manycore NICs provide performant mechanisms to isolate competing tenants. Further, performance on manycore NICs can degrade significantly if the working set does not fit within the core's cache.
- **RMT NICs** use on-NIC reconfigurable match+action (RMT) pipeline to implement NIC offloads. The types of offloads that can be supported by RMT pipelines are limited because each pipeline stage must be able to handle processing a new packet every single clock cycle.

PANIC: A High-Performance Programmable NICfor Multi-tenant Networks

Problems/Challenges

- Existing programmable NIC designs, categorized below, have key limitations:
  - Pipeline-of-Offloads (ASIC + FPGA)
    - Modifying chaining requires significant amount of time and developer effort for FPGA synthesis
    - Slow offloads cause packet loss or head-of-line (HOL) blocking
  - Manycore NICs (CPUs)
    - CPU cores add tens of microseconds of additional latency
    - No performant mechanisms today to isolate competing tenants
    - Performance degrades significantly if working set done not fit in cache
  - RMT NICs (programmable ASIC)
    - Limited offload support
    - Each pipeline stage much be able to handle processing a new packet every single clock cycle

ARTIFACT EVALUATED usenix AVAILABLE | ARTIFACT EVALUATED usenix FUNCTIONAL | ARTIFACT EVALUATED usenix REPRODUCED

PANIC: A High-Performance Programmable NIC for Multi-tenant Networks

Jiaxin Lin, *University of Wisconsin-Madison*
Kiran Patel, *University of Illinois at Chicago*
Brent E. Stephens, *University of Illinois at Chicago*
Anirudh Sivaraman, *New York University (NYU)*
Aditya Akella, *University of Wisconsin-Madison*

Abstract

Programmable NICs have diverse uses, and there is a need for a NIC platform that can offload computation from multiple co-resident applications to many different types of substrates, including hardware accelerators, embedded FPGAs, and embedded processor cores. Unfortunately, there is no existing NIC design that can simultaneously support a large number of diverse offloads while ensuring high throughput/low latency, multi-tenant isolation, flexible offload chaining, and support for offloads with variable performance.

This paper presents PANIC, a new programmable NIC. There are two new key components of the PANIC design that enable it to overcome the limitations of existing NICs: 1) A high-performance switching interconnect that scalably connects independent engines into offload chains, and 2) A new hybrid push/pull packet scheduler that provides cross-tenant performance isolation and low-latency load-balancing across parallel offload engines. From experiments performed on an 100 Gbps FPGA-based prototype, we find that this design overcomes the limitations of state-of-the-art programmable NICs.

1 Introduction

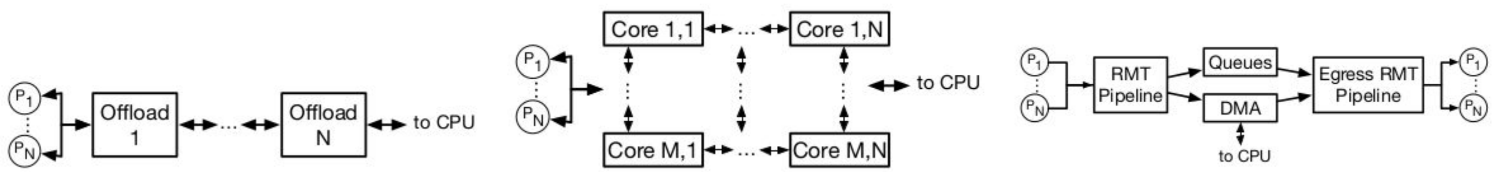
The gap between network line-rates and the rate at which a CPU can produce and consume data is widening rapidly [71, 66]. Emerging *programmable ("smart") NICs* can help overcome this problem [32]. There are many different types of offloads that can be implemented on a programmable NIC. These offloads, which accelerate computation across all of the different layers of the network stack, can reduce load on the general purpose CPU, reduce latency, and increase throughput [32, 48, 59, 69, 13].

Many different cloud and datacenter applications and use cases have been shown to benefit from offloading computation to programmable NICs [13, 48, 59, 42, 32, 37, 49, 46, 62, 47, 30, 36, 70, 69, 35, 55, 45]. However, there is no single … offload that can improve performance in all … we anticipate that different applications will …n chains of offloads, and that the operator will …se chains with infrastructure-related offloads …on her programmable NICs. To realize this …per presents PANIC, a new scalable and high- …rogrammable NIC for multi-tenant networks that supports a wide variety of different types of offloads and composes them into isolated offload chains.

To enable cloud operators to provide NIC offload chains as a service to tenants, a programmable NIC must support: 1) Offload variety: some offloads like cryptography are best suited for hardware implementations, while an offload providing a low-latency bypass for RPCs in an application is better suited for an embedded core [51]; 2) Offload chaining: to minimize wasted chip area on redundant functions, the NIC should facilitate composing independent hardware offload units into a chain as needed, with commonly-needed offloads shared across tenants; 3) Multi-tenant isolation: tenants should not be able to consume more than their allocation of a shared offload; 4) Variable-performance offloads: there are useful offloads that are not guaranteed to run at line-rate, as well as important offloads that run with low latency and at line-rate.

There exist many different programmable NICs [32, 12, 75, 31, 58, 72, 23, 24, 11, 57, 53, 54, 52, 76], but, there is no programmable NIC that is currently able to provide all of the above properties. Existing NIC designs can be categorized as follows, with each category imposing key limitations:

- **Pipeline-of-Offloads NICs** place multiple offloads in a pipeline to enable packets to be processed by a chain of functions [52, 32]. Chaining can be modified in these NICs today but requires a significant amount of time and developer effort for FPGA synthesis, and slow offloads cause packet loss or head-of-line (HOL) blocking.
- **Manycore NICs** load balance packets across many embedded CPU cores, with the CPU core then controlling the processing of packets as needed for different offloads [23, 24, 53, 54, 57, 72, 58]. These designs suffer from performance issues because embedded CPU cores add tens of microseconds of additional latency [32]. Also, no existing manycore NICs provide performant mechanisms to isolate competing tenants. Further, performance on manycore NICs can degrade significantly if the working set does not fit within the core's cache.
- **RMT NICs** use on-NIC reconfigurable match+action (RMT) pipeline to implement NIC offloads. The types of offloads that can be supported by RMT pipelines are limited because each pipeline stage must be able to handle processing a new packet every single clock cycle.



(a) A pipeline-of-offloads architecture (b) A tiled manycore NIC architecture (c) A NIC with RMT pipeline [42]

Figure 1: Illustrations of existing programmable NIC architectures.

Red Hat

PANIC: A High-Performance Programmable NICfor Multi-tenant Networks

Background and related work (Sections 2 and 8)

- NICs should support both hardware and software offloads since not all offloads are best implemented on the same type of underlying engine. For example, crypto offload works better using hardware accelerators while walking a hash table resident in main memory is better suited for embedded cores.
- Applications, and even individual packets, can have different requirements. Secure remote memory access may require: crypto + congestion control + RDMA offload blocks. Key value store - that serves requests both from within data center and WAN distributed clients - can require an IPSec and/or compression offloads, but only WAN packets will likely use them.
- Some offloads may not run at line-rate. Of the compression, cryptography, authentication, and inference offloads that we ran on hardware, only inference was able to run at 100 Gbps. Compression and authentication performance depends on packet size. Slow offloads can be duplicated across multiple engines (e.g., 3 AES-256 engines) for line-rate operation.
- An offload that is used for TX and RX on a dual port NIC needs to operate at four times line-rate to prevent becoming a bottleneck.

NIC Design	Offload Chaining	Multi-Tenant Isolation	Variable Perf	High Perf	Offload Variety
Pipeline	✗	✗	✗	✓	✗
Manycore	✓	✗	✓	✗	✓
RMT	✗	✓	✗	✓	✗

Table 2: **Programmable NIC designs compared w.r.t. the requirements in Section 2.1.**

This paper presents the design, implementation and evaluation of PANIC, a new NIC that overcomes the key limitations of existing NIC designs. PANIC draws inspiration from recent work on reconfigurable (RMT) switches [21, 67, 68, 27, 16]. PANIC's design leverages three key principles:

1. *Offloads should be self-contained.* The set of potentially useful offloads is diverse and vast, spanning all of the layers of the network stack. As such, a programmable NIC should be able to support both hardware IP cores and embedded CPUs as offloads.
2. *Packet scheduling, buffering, and load-balancing should be centralized* for the best performance and efficiency because decentralized decisions and per-offload queuing can lead to poor tail response latencies and poor buffer utilization due to load imbalances.
3. Because the cost of small/medium-sized non-blocking fabrics is small relative to the NIC overall, *the offloads should be connected by a non-blocking/low-oversubscribed switching fabric* to enable flexible chaining of offloads.

Following these design principles, this paper makes three key contributions: 1) A novel programmable NIC design where diverse offloads are connected to a non-blocking switching fabric, with chains orchestrated by a programmable RMT pipeline, 2) A new hybrid push/pull scheduler-and-load balancer with priority-aware packet dropping, and 3) An analysis of the costs of on-NIC programmable switching and scheduling that finds them to be low relative to the NIC as a whole.

The PANIC NIC has four components: 1) an RMT switch pipeline, 2) a switching fabric, 3) a central scheduler, and 4) self-contained compute units. The RMT pipeline provides programmable chain orchestration. A high performance interconnect enables programmable chaining at line-rate. The central scheduler provides isolation, buffer management, and load-balancing. Self-contained compute units may be either hardware accelerators or embedded cores and are not required to run at line-rate.

To evaluate the feasibility of PANIC, we have performed both ASIC analysis and experiments with an FPGA prototype. Our ASIC analysis demonstrates the feasibility of the PANIC architecture and shows that the crossbar interconnect topology scales well up to 32 total attached compute units. Our FPGA prototype can perform dynamic offload chaining at 100 Gbps, and achieves nanosecond-level ($<0.8\,\mu s$) packet scheduling and load-balancing under a variety of chaining configurations. We empirically show that PANIC can handle multi-tenant isolation and below line-rate offloads better than a state-of-the-art pipeline-based design. Our end-to-end experiments in a small scale testbed demonstrate that PANIC can achieve dynamic bandwidth allocation and prioritized packet scheduling at 100 Gbps. In total, the components of PANIC, which includes an 8 * 8 crossbar, only consume a total of 11.27% of the total logic area (LUTs) available on the Xilinx UltraScale Plus FPGA that we used. The Verilog code for our FPGA prototype is publicly available [1].

Offload	Config	Tput (Gbps)	Delay
Data Processing			
Compression (Lzrw1)	HW@300MHz	3.6	0.05-3.3μs
Cryptography (AES-256)	HW@300MHz	38.4	407ns
Cryptography (AES-256)	CPU@1.5GHz	0.154	–
Network Processing			
Authentication (SHA1)	HW@220MHz	113.0	0.47-10.8μs
Authentication (SHA1)	CPU@1.5GHz	0.192	–
Application Processing			
Inference (3-layer-NN)	HW@200MHz	120	66ns

Table 1: **A breakdown of the performance of different offloads when implemented in either hardware or software.**

2 Motivation

We discuss in detail the requirements that we envision programmable NICs in multi-tenant networks ought to meet. We then explain why existing NICs designs fail to meet them.

2.1 Requirements

1. Offload Variety: There are a large variety of network offloads, and different types of offloads have different needs. Not all offloads are best implemented on the same type of underlying engine. For example, a cryptography offload can provide much better performance if implemented with a hardware accelerator built from a custom IP core instead of an embedded processor core. To shed light on this, we experimented with a few different types of offloads using an Alpha Data ADM-PCIE-9V3 Programmable NIC [12] to evaluate the behavior of different hardware IP cores that could be used as on-NIC accelerators, and the Rocket Chip Generator [14] to perform cycle-accurate performance measurements of a RISC V CPU to understand the costs of running these offload with an on-NIC embedded processor. Our results in Table 1 indeed show that offloads for encryption/decryption and authentication are a poor fit for embedded CPU designs and should be implemented in hardware.

In contrast, an application-specific offload to walk a hash table that is resident in main memory is better suited for an embedded processor core because a hardware offload may not provide enough flexibility [51]. Thus, a programmable NIC should ideally provide support for both hardware and software offloads.

2. Dynamic Offload Chaining: In the case of hardware accelerators, it is important to be able to compose independent offload functionality into a chain/pipeline to avoid wasted area on redundant functionality. For example, using a programmable NIC to implement a secure remote memory access for a tenant may require the tenant to compose cryptography, congestion control, and RDMA offload engines.

[1] PANIC artifact: https://bitbucket.org/uw-madison-networking-research/panic_osdi20_artifact

PANIC: A High-Performance Programmable NICfor Multi-tenant Networks

Architecture overview	Operational overview, Offload variety support, Dynamic offload chaining support, Policies for dynamic multi-tenant isolation, Support for offloads with variable and below line-rate performance, Support for high performance
Design of individual components	RMT pipelines, High performance interconnect, Centralized scheduler, Compute unit
ASIC analysis	RMT, PIFO parser, Interconnect, Compute units
FPGA Prototype	RMT pipelines, FPGA-based crossbar, Central scheduler and packet buffer, Compute units,
Evaluation	Testbed and methodology, Microbenchmarks, Comparison with the pipeline design, RISCV core performance, Hardware resource usage, End-to-end performance

Paper 3: Security Track

Orchard: Differentially Private Analytics at Scale

Orchard: Differentially Private Analytics at Scale

“Apple pie”

- When operating a large distributed system, it is often useful to collect some data from the users’ devices—e.g., to train models that will help to improve the system.

- Since this data is often sensitive, differential privacy is an attractive choice, and several deployed systems are using it today to protect the privacy of their users.
  - Google is using differential privacy to monitor the Chrome web browser.
  - Apple is using it in iOS and macOS, e.g., to train its models for predictive typing and to identify apps with high energy or memory usage.
  - Other deployments include those at Microsoft and at Snap.

- Today, this data is typically collected using local differential privacy.
  - Each user device individually adds some random noise to its own data.
  - Then each user uploads the data to a central entity.
  - The central entity then aggregates the uploads and delivers the final result.

- Local differential privacy can be done efficiently at scale.

ARTIFACT EVALUATED usenix AVAILABLE | ARTIFACT EVALUATED usenix FUNCTIONAL | ARTIFACT EVALUATED usenix REPRODUCED

Orchard: Differentially Private Analytics at Scale

Edo Roth, Hengchu Zhang, Andreas Haeberlen, Benjamin C. Pierce

University of Pennsylvania

Abstract

This paper presents Orchard, a system that can answer queries about sensitive data that is held by millions of user devices, with strong differential privacy guarantees. Orchard combines high accuracy with good scalability, and it uses only a single untrusted party to facilitate the query. Moreover, whereas previous solutions that shared these properties were custom-built for specific queries, Orchard is general and can accept a wide range of queries. Orchard accomplishes this by rewriting queries into a distributed protocol that can be executed efficiently at scale, using cryptographic primitives.

Our prototype of Orchard can execute 14 out of 17 queries chosen from the literature; to our knowledge, no other system can handle more than one of them in this setting. And the costs are moderate: each user device typically needs only a few megabytes of traffic and a few minutes of computation time. Orchard also includes a novel defense against malicious users who attempt to distort the results of a query.

1 Introduction

When operating a large distributed system, it is often useful to collect some data from the users' devices—e.g., to train models that will help to improve the system. Since this data is often sensitive, differential privacy [28] is an attractive choice, and several deployed systems are using it today to protect the privacy of their users. For instance, Google is using differential privacy to monitor the Chrome web browser [31], and Apple is using it in iOS and macOS, e.g., to train its models for predictive typing and to identify apps with high energy or memory usage [7,8]. Other deployments exist, e.g., at Microsoft [27] and at Snap [68].

Today, this data is typically collected using *local* differential privacy [31]: each user device *individually* adds some random noise to its own data and then uploads it to a central entity, which aggregates the uploads and delivers the final result. This can be done efficiently at scale, but the final result contains an enormous amount of noise: as Google notes [14], even in a deployment with a billion users, it is easy to miss signals from a million users. Utility can be improved by reducing the amount of noise, but this weakens the privacy guarantee considerably, to the point where it becomes almost meaningless [80].

One way to avoid this problem is to collect the data using *global* differential privacy instead. In this approach, each device provides its raw, un-noised data to the central aggregator, which then adds random noise only once. This clearly produces results that are more precise, but it also requires a lot of trust in the aggregator, who now receives the individual users' raw data and must be trusted not to look at it. Cryptographic techniques like multiparty computation [84] and fully homomorphic encryption [38] could theoretically avoid this problem, but, at least with current technology, scaling either approach to millions of participants seems implausible.

The recently proposed Honeycrisp system [76] can provide global differential privacy at scale, with a single, untrusted aggregator. Instead of fully homomorphic encryption, Honeycrisp uses additively homomorphic encryption, which is much more efficient. However, the price to pay is that Honeycrisp can answer only one specific query, namely count-mean sketches [8] with additional use of the sparse-vector operator. This query does have important applications (for instance, it is used in Apple's iOS), but it is by no means the *only* query one might wish to ask: the literature is full of other interesting queries that can be performed with global differential privacy (e.g., [15,31,40,41,55,64,70,83]). Right now, we are not aware of any systems that can answer even one of these queries at scale, using only a single, untrusted aggregator.

In this paper, we show how to substantially expand the variety of queries that can be answered efficiently in this highly distributed setting. Our key insight is that many differentially private queries have a lot more in common than at first meets the eye: while most of them transform, group, or otherwise process the input data in some complicated way, the heart of the algorithm is (almost) always a sequence of sums, each computed over some values that are derived from the users' input data. This happens to be exactly the kind of computation that Honeycrisp's collect-and-test (CaT) primitive can perform efficiently, using additively homomorphic encryption. Thus, CaT turns out to be far more general than it may seem: it can perform the distributed parts of many queries, leaving only a few smaller computations that can safely be done by the aggregator, or locally on each user device.

The key challenge is that, for many queries, the connection to sums over per-user data is far from obvious. Many differentially private queries were designed for a centralized setting where the aggregator has an unencrypted data set and

USENIX Association 14th USENIX Symposium on Operating Systems Design and Implementation 1065

Orchard: Differentially Private Analytics at Scale

Problems/Challenges

- The final result of local differential privacy contains an enormous amount of noise.
  - Even in a deployment with a billion users, it is easy to miss signals from a million users.
  - Reducing noise weakens privacy guarantee considerably.
- Global differential privacy can address this since noise is only added once i.e. by the aggregator.
- However, global differential privacy requires a lot more trust in the aggregator since individual users have to send raw data and trust that the aggregator will not look at it.
- Crypto techniques like Multi Party Computation and Fully Homomorphic Encryption can avoid the untrusted aggregator problem, but do not scale to millions of participants with current technology.
- Systems like Honeycrisp use additively homomorphic encryption which is much more efficient at scaling, but can only answer the count-mean sketches query.

ARTIFACT EVALUATED usenix AVAILABLE | ARTIFACT EVALUATED usenix FUNCTIONAL | ARTIFACT EVALUATED usenix REPRODUCED

Orchard: Differentially Private Analytics at Scale

Edo Roth, Hengchu Zhang, Andreas Haeberlen, Benjamin C. Pierce

University of Pennsylvania

Abstract

This paper presents Orchard, a system that can answer queries about sensitive data that is held by millions of user devices, with strong differential privacy guarantees. Orchard combines high accuracy with good scalability, and it uses only a single untrusted party to facilitate the query. Moreover, whereas previous solutions that shared these properties were custom-built for specific queries, Orchard is general and can accept a wide range of queries. Orchard accomplishes this by rewriting queries into a distributed protocol that can be executed efficiently at scale, using cryptographic primitives.

Our prototype of Orchard can execute 14 out of 17 queries chosen from the literature; to our knowledge, no other system can handle more than one of them in this setting. And the costs are moderate: each user device typically needs only a few megabytes of traffic and a few minutes of computation time. Orchard also includes a novel defense against malicious users who attempt to distort the results of a query.

1 Introduction

When operating a large distributed system, it is often useful to collect some data from the users' devices—e.g., to train models that will help to improve the system. Since this data is often sensitive, differential privacy [28] is an attractive choice, and several deployed systems are using it today to protect the privacy of their users. For instance, Google is using differential privacy to monitor the Chrome web browser [31], and Apple is using it in iOS and macOS, e.g., to train its models for predictive typing and to identify apps with high energy or memory usage [7,8]. Other deployments exist, e.g., at Microsoft [27] and at Snap [68].

Today, this data is typically collected using *local* differential privacy [31]: each user device *individually* adds some random noise to its own data and then uploads it to a central entity, which aggregates the uploads and delivers the final result. This can be done efficiently at scale, but the final result contains an enormous amount of noise: as Google notes [14], even in a deployment with a billion users, it is easy to miss signals from a million users. Utility can be improved by reducing the amount of noise, but this weakens the privacy guarantee considerably, to the point where it becomes almost meaningless [80].

One way to avoid this problem is to collect the data using *global* differential privacy instead. In this approach, each device provides its raw, un-noised data to the central aggregator, which then adds random noise only once. This clearly produces results that are more precise, but it also requires a lot of trust in the aggregator, who now receives the individual users' raw data and must be trusted not to look at it. Cryptographic techniques like multiparty computation [84] and fully homomorphic encryption [38] could theoretically avoid this problem, but, at least with current technology, scaling either approach to millions of participants seems implausible.

The recently proposed Honeycrisp system [76] can provide global differential privacy at scale, with a single, untrusted aggregator. Instead of fully homomorphic encryption, Honeycrisp uses additively homomorphic encryption, which is much more efficient. However, the price to pay is that Honeycrisp can answer only one specific query, namely count-mean sketches [8] with additional use of the sparse-vector operator. This query does have important applications (for instance, it is used in Apple's iOS), but it is by no means the *only* query one might wish to ask: the literature is full of other interesting queries that can be performed with global differential privacy (e.g., [15,31,40,41,55,64,70,83]). Right now, we are not aware of any systems that can answer even one of these queries at scale, using only a single, untrusted aggregator.

In this paper, we show how to substantially expand the variety of queries that can be answered efficiently in this highly distributed setting. Our key insight is that many differentially private queries have a lot more in common than at first meets the eye: while most of them transform, group, or otherwise process the input data in some complicated way, the heart of the algorithm is (almost) always a sequence of sums, each computed over some values that are derived from the users' input data. This happens to be exactly the kind of computation that Honeycrisp's collect-and-test (CaT) primitive can perform efficiently, using additively homomorphic encryption. Thus, CaT turns out to be far more general than it may seem: it can perform the distributed parts of many queries, leaving only a few smaller computations that can safely be done by the aggregator, or locally on each user device.

The key challenge is that, for many queries, the connection to sums over per-user data is far from obvious. Many differentially private queries were designed for a centralized setting where the aggregator has an unencrypted data set and

Red Hat

Orchard: Differentially Private Analytics at Scale

Background and related work (Section 2, 8)

- Important goals for a differential privacy system:
  - Privacy
    - The amount of information that either the aggregator or other users can learn about the private data of an honest user should be bounded, according to the formulation of differential privacy.
  - Correctness
    - If all users are honest, the answers to queries should be drawn from a distribution that is centered on the correct answer and has a known shape.
  - Robustness
    - Malicious users should not be able to significantly distort the answers.
  - Efficiency
    - Most users should not need to contribute more than a few MB of bandwidth and a few seconds of computation time per query.

can perform arbitrary computations on it. Such queries often need to be transformed substantially, and existing operators need to be broken down into their constituents, in order to expose the internal sums. Moreover, a naïve transformation can result in a very large number of sums—often far more than are strictly necessary. Thus, optimizations are needed to maintain efficiency.

We present a system called Orchard that can automatically perform these steps for a large variety of queries. Orchard accepts centralized queries written in an existing query language, transforms them into distributed queries that can be answered at scale, and then executes these queries using a generalization of the CaT mechanism from Honeycrisp. Among 17 queries we collected from the literature, Orchard was able to execute 14; the others are not a good fit for our highly distributed setting and would require a different approach.

Our experimental evaluation of Orchard shows that most queries can be answered efficiently: with 1.3 billion users (roughly the size of Apple's macOS/iOS deployment [6]), most user devices would need only a few megabytes of traffic and a few minutes of computation time, while the aggregator would need about 900 cores to get the answer within one hour. For queries that make use of the sparse-vector operator, this is competitive with Honeycrisp; for the other queries we consider, we are not aware of any other approach that is practical in this setting. In summary, our contributions are:

- the observation that many differentially private queries can be transformed into a sequence of noised sums (Section 2);
- a simple language for writing queries (Section 3);
- a transformation of queries in this language to protocols that can answer them in a distributed setting, using only a single, untrusted aggregator (Section 4);
- the design of Orchard, a platform that can efficiently execute the transformed queries (Section 5);
- a prototype implementation of Orchard (Section 6); and
- an experimental evaluation (Section 7).

We discuss related work in Section 8 and conclude the paper in Section 9.

2 Overview

Scenario: We consider a scenario—illustrated in Figure 1—with a very large number of users (millions), who each hold some sensitive data, and a central entity, the *aggregator*, that wishes to answer queries about this data. We assume that each user has a device (say, a cell phone or a laptop) that can perform some limited computations, while the aggregator has access to substantial bandwidth and computation power (say, a data center).

Threat model: We make the OB+MC assumption from [76]—that is, we assume that the aggregator is honest-but-curious

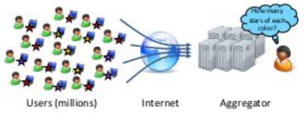


Figure 1: Scenario.

(HbC) when the system is first deployed and usually remains HbC thereafter, but may occasionally be Byzantine (OB) for limited time periods; for instance, the aggregator could be a large company that is under public scrutiny and would not violate privacy systematically, but may have a rogue employee who might tamper with the system and not be discovered immediately. For the users, we assume that most of them are correct (MC) but that a small percentage—say, 2–3%—can be Byzantine at any given time. This is different from the typical assumption in the BFT literature, where one often assumes that up to a third, or even half, of the nodes can be Byzantine. However, BFT systems are typically a lot smaller than the systems we consider: with 4–7 replicas, compromising a third of the systems means just one or two nodes, whereas, in Apple's deployment with 1.3 billion users, a 3% bound would mean 39 million malicious users, which is much larger than, e.g., a typical botnet.

Assumptions: Our key assumptions are (1) that the approximate number of users is known and (2) that the adversary cannot create and collude with a nontrivial number of Sybils. For instance, the devices could have hardware support for secure identities, such as Apple's T2 chip or Intel's SGX.

Goals: We have four key goals for Orchard:

- **Privacy:** The amount of information that either the aggregator or other users can learn about the private data of an honest user should be bounded, according to the formulation of differential privacy.
- **Correctness:** If all users are honest, the answers to queries should be drawn from a distribution that is centered on the correct answer and has a known shape;
- **Robustness:** Malicious users should not be able to significantly distort the answers; and
- **Efficiency:** Most users should not need to contribute more than a few MB of bandwidth and a few seconds of computation time per query.

2.1 Differential privacy

Differential privacy [28] is a property of randomized queries that take a database as input and return an aggregate output. Informally, a query is differentially private if changing any single row in the input database results in "almost no change" in the output. If each row represents the data of a single individual, this means that any single individual has a statistically

Orchard: Differentially Private Analytics at Scale

Background and related work (Section 2, 8)

- Differential privacy is a property of randomized queries that take a database as input and return an aggregate output. Informally, a query is differentially private if changing any single row in the input database results in “almost no change” in the output.

- If each row represents the data of a single individual, this means that any single individual has a statistically negligible effect on the output. This guarantee is quantified in the form of a parameter, ε, which controls how much the output can vary based on changes to a single row.

- A standard method for achieving differential privacy for numeric queries is the Laplace mechanism, which involves two steps:
    - calculating the sensitivity, *s*, of the query which is how much the un-noised output can change based on a change to a single row
    - adding noise drawn from a Laplace distribution with scale parameter s/ε; this results in ε-differential privacy.

- For queries with discrete values, the standard method is exponential mechanism which is based on:
    - A “quality score” that measures how well a value ‘x’ represents a database ‘d’
    - The sensitive of the quality score.

- Differential privacy is compositional - if we evaluate two queries which are ε1 and ε2 differential private, then publishing results from both queries is at most (ε1 + ε2) differentially private.

- We can define a privacy budget (εmax) that corresponds to the maximum acceptable privacy loss.
    - The ε for each query is deducted from this budget till it is exhausted.

can perform arbitrary computations on it. Such queries often need to be transformed substantially, and existing operators need to be broken down into their constituents, in order to expose the internal sums. Moreover, a naïve transformation can result in a very large number of sums—often far more than are strictly necessary. Thus, optimizations are needed to maintain efficiency.

We present a system called Orchard that can automatically perform these steps for a large variety of queries. Orchard accepts centralized queries written in an existing query language, transforms them into distributed queries that can be answered at scale, and then executes these queries using a generalization of the CaT mechanism from Honeycrisp. Among 17 queries we collected from the literature, Orchard was able to execute 14; the others are not a good fit for our highly distributed setting and would require a different approach.

Our experimental evaluation of Orchard shows that most queries can be answered efficiently: with 1.3 billion users (roughly the size of Apple's macOS/iOS deployment [6]), most user devices would need only a few megabytes of traffic and a few minutes of computation time, while the aggregator would need about 900 cores to get the answer within one hour. For queries that make use of the sparse-vector operator, this is competitive with Honeycrisp; for the other queries we consider, we are not aware of any other approach that is practical in this setting. In summary, our contributions are:

- the observation that many differentially private queries can be transformed into a sequence of noised sums (Section 2);
- a simple language for writing queries (Section 3);
- a transformation of queries in this language to protocols that can answer them in a distributed setting, using only a single, untrusted aggregator (Section 4);
- the design of Orchard, a platform that can efficiently execute the transformed queries (Section 5);
- a prototype implementation of Orchard (Section 6); and
- an experimental evaluation (Section 7).

We discuss related work in Section 8 and conclude the paper in Section 9.

2 Overview

Scenario: We consider a scenario—illustrated in Figure 1—with a very large number of users (millions), who each hold some sensitive data, and a central entity, the *aggregator*, that wishes to answer queries about this data. We assume that each user has a device (say, a cell phone or a laptop) that can perform some limited computations, while the aggregator has access to substantial bandwidth and computation power (say, a data center).

Threat model: We make the OB+MC assumption from [76]—that is, we assume that the aggregator is honest-but-curious

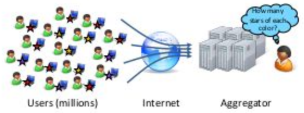


Figure 1: Scenario.

(HbC) when the system is first deployed and usually remains HbC thereafter, but may occasionally be Byzantine (OB) for limited time periods; for instance, the aggregator could be a large company that is under public scrutiny and would not violate privacy systematically, but may have a rogue employee who might tamper with the system and not be discovered immediately. For the users, we assume that most of them are correct (MC) but that a small percentage—say, 2–3%—can be Byzantine at any given time. This is different from the typical assumption in the BFT literature, where one often assumes that up to a third, or even half, of the nodes can be Byzantine. However, BFT systems are typically a lot smaller than the systems we consider: with 4–7 replicas, compromising a third of the systems means just one or two nodes, whereas, in Apple's deployment with 1.3 billion users, a 3% bound would mean 39 million malicious users, which is much larger than, e.g., a typical botnet.

Assumptions: Our key assumptions are (1) that the approximate number of users is known and (2) that the adversary cannot create and collude with a nontrivial number of Sybils. For instance, the devices could have hardware support for secure identities, such as Apple's T2 chip or Intel's SGX.

Goals: We have four key goals for Orchard:

- **Privacy:** The amount of information that either the aggregator or other users can learn about the private data of an honest user should be bounded, according to the formulation of differential privacy.
- **Correctness:** If all users are honest, the answers to queries should be drawn from a distribution that is centered on the correct answer and has a known shape;
- **Robustness:** Malicious users should not be able to significantly distort the answers; and
- **Efficiency:** Most users should not need to contribute more than a few MB of bandwidth and a few seconds of computation time per query.

2.1 Differential privacy

Differential privacy [28] is a property of randomized queries that take a database as input and return an aggregate output. Informally, a query is differentially private if changing any single row in the input database results in “almost no change” in the output. If each row represents the data of a single individual, this means that any single individual has a statistically

Orchard: Differentially Private Analytics at Scale

Programming language selection: Fuzz	Running example: k-means, Language features, Alternative languages
Transform centralized Fuzz queries to support distributed execution	Program zones, The *bmcs* operator. Extracting dependencies, Transformation to *bmcs* form, Optimizations, Limitations
Distributed query execution	Overall workflow, Security: Aggregator, Security: Malicious clients, Handling churn
Implementation	Encryption, MPC, Secret sharing, Verifiable computation, Security parameters
Evaluation	Coverage, Optimizations, Robustness to malicious users, Experimental setup, Cost for normal participants, Cost for the committee, Cost for the aggregator

Red Hat

Join us for the next session!

Session 1	4/3/2021	A perspective on research papers
Session 2	5/4/2021	Identifying worthwhile papers
Session 3	6/1/2021	Discussing research papers

Sign-up / Comments / Suggestions / Feedback

https://forms.gle/6Y2ZBH2Bq2y5Qmie7

Thank you!