DVM: Towards a Datacenter-Scale Virtual Machine

Zhiqiang Ma Zhonghua Sheng Lin Gu

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong
{zma,szh,lingu}@cse.ust.hk

Liufei Wen Gong Zhang

Huawei Technologies
Shenzhen, China
{wenliufei,nicholas}@huawei.com

Abstract

As cloud-based computation becomes increasingly important, providing a general computational interface to support datacenter-scale programming has become an imperative research agenda. Many cloud systems use existing virtual machine monitor (VMM) technologies, such as Xen, VMware, and Windows Hypervisor, to multiplex a physical host into multiple virtual hosts and isolate computation on the shared cluster platform. However, traditional multiplexing VMMs do not scale beyond one single physical host, and it alone cannot provide the programming interface and cluster-wide computation that a datacenter system requires. We design a new instruction set architecture, DISA, to unify myriads of compute nodes to form a big virtual machine called DVM, and present programmers the view of a single computer where thousands of tasks run concurrently in a large, unified, and snapshotted memory space. The DVM provides a simple yet scalable programming model and mitigates the scalability bottleneck of traditional distributed shared memory systems. Along with an efficient execution engine, the capacity of a DVM can scale up to support large clusters. We have implemented and tested DVM on three platforms, and our evaluation shows that DVM has excellent performance in terms of execution time and speedup. On one physical host, the system overhead of DVM is comparable to that of traditional VMMs. On 16 physical hosts, the DVM runs 10 times faster than MapReduce/Hadoop and X10. On 256 EC2 instances, DVM shows linear speedup on a parallelizable workload.

Categories and Subject Descriptors D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming—Parallel programming; C.2.4 [*COMPUTER-COMMUNICATION NETWORKS*]: Distributed Systems

General Terms Design, Performance, Experimentation

Keywords Datacenter, Virtualization, Cloud Computing

1. Introduction

Emerging as a paradigm shifting technology, cloud-based computation has demonstrated tremendous capability in web applications and a wide range of computational tasks, such as business intelligence, bioinformatics, computational finance, chemical analysis, and environmental sciences [17, 18, 21, 42, 43], especially for processing large data sets [10, 20, 27]. Cloud-based systems rely on abundantly provisioned datacenters strategically distributed in geographic regions [7, 24]. A datacenter usually contains tens of thousands of compute servers made of commodity hardware, employs high-bandwidth switching networks, and often utilizes virtualization and distributed services to manage resources and provide a scalable computing platform [1, 2, 10].

It is believed that virtualization is a fundamental component in cloud technology. Particularly, ISA-level virtual machine monitors (VMM) are used by major cloud providers for packaging resources and enforcing isolation [1, 3], sometimes providing underlying support for higher-level language constructs [2]. However, the traditional VMMs are typically designed to work on a single physical host and multiplex it to be several virtual machine instances [8, 30, 46]. Though it is possible to build management or single system image (SSI) services to coordinate multiple physical or virtual hosts [12, 38, 46], they fall short of providing the functionality and abstraction that allow users to develop and execute programs as if the underlying platform were a single big "computer" [9]. Extending traditional ISA abstractions beyond a single machine has only met limited success. For example, vNUMA extends the IA-64 (Itanium) instruction set to a cluster of machines [14], but finds it necessary to simplify the memory operation semantics and shows difficulty in scaling to very large clusters.

On the other hand, exposing the distributed detail of the platform to the application layer leads to increased complexity in programming, decreased performance, and, sometimes, loss of generality. In recent years, application frameworks [20, 27] and productivity-oriented parallel programming languages [4, 15, 40, 48] have been widely used in cluster-wide computation. For example, MapReduce and its open-source variant, Hadoop, are perhaps the most widely-used application framework in the datacenter environment [34]. Without completely hiding the distributed hardware, MapReduce requires programmers to partition the program state so that each map and reduce task can be executed on one individual host, and enforces a specific control flow and dependence relation to ensure correctness and reliability. This leads to a restricted programming model which is efficient for "embarrassingly parallel" programs, where data can be partitioned with minimum dependence and thus the processing is easily parallelizable, but makes it difficult to design sophisticated and general applications [21, 34, 41, 47]. The language-level solutions, such as X10 [15], Chapel [13] and Fortress [4], are more general and give programmers better control over the program logic flows, parallelization and synchronization, but the static locality, programmer-specified synchronization, and the linguistic artifacts influence such solutions to perform well with one set of programs but fail to deliver performance, functionality, or ease-of-programming with another. It remains an open problem to design an effective system architec-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VEE'12, March 3–4, 2012, London, England, UK.

Copyright © 2012 ACM 978-1-4503-1175-5/12/03. . . $10.00

ture and programming abstraction to support general, flexible, and concurrent application workloads with sophisticated processing.

Considering that ISAs indeed provide a clearly defined interface between hardware and software and allow both layers to evolve and diversify, we believe it is effective to unify the computers in a datacenter at the instruction level, and present the programmers the view of a big virtual machine that can potentially scale to the entire datacenter. The key is to overcome the scalability bottleneck in traditional instruction sets and memory systems. Hence, instead of porting an existing ISA, we design a new ISA, Datacenter Instruction Set Architecture (DISA), for datacenter-scale computation. Through its instruction set, memory model and parallelization mechanism, the DISA architecture presents the programmers an abstraction of a large-scale virtual machine, called the DISA Virtual Machine (DVM), which runs on a plurality of physical hosts inside a datacenter. It defines an interface between the DVM and the software running above it, and ensures that a program can execute on any collection of computers that implement DISA.

Different from existing VMMs (e.g. VMware [46], Xen [8], vNUMA [14]) on traditional architectures (e.g., x86), DVM and DISA make design choices on the instruction set, execution engine, memory semantics and dependence resolution to facilitate scalable computation in a dynamic environment with many loosely coupled physical or virtual hosts. There are certainly many technical challenges in creating an ISA scalable to a union of thousands of computers, and backward compatibility used to constrain the application of new ISAs. Fortunately, the cloud computing paradigm presents opportunities for innovations at this basic system level—first, the datacenter is often a controlled environment where the owner can autonomously decide internal technical specifications; second, major cloud-based programming interfaces (e.g., RESTful APIs, RPC, streaming) require minimum instruction-level compatibility; finally, there is only a manageable set of "legacy code". The goals of our design of DVM are:

- **General**. DVM should support the design and execution of general programs in datacenters.
- **Efficient**. DVM should be efficient and deliver high performance.
- **Scalable**. DVM should be scalable to run on a plurality of hosts, execute a large number of tasks, and handle large datasets.
- **Portable**. DVM should be portable across different clusters with different hardware configurations, networks, etc.
- **Simple**. It should be easy to program the DVM; DISA should be simple to implement.

With the design goals stated, we revisit the approach and rationale of unifying the computers in a datacenter at the ISA layer. The programming framework, language, and system call/API are also possible abstraction levels for large-scale computation. However, as aforementioned, the frameworks and languages have their limitations and constraints. Hence, they fall short of providing the required *generality* and *efficiency*. Using system calls/APIs to invoke distributed system services can be *efficient* and reasonably *portable*. However, an abstraction on this level creates dichotomized programming domains and requires the programmers to explicitly handle the complexity of organizing the program logic around the system calls/APIs and filling any semantic gap, which fails to provide the illusion of a "single big machine" to the programmers and leads to a far less *general* and *easy-to-program* approach than what we aim to. The existing ISAs, designed for a single computer with up to a moderate number of cores, can hardly provide the *scalability* required by the "machine" that scales to the datacenter size. On the other hand, the design of the DVM is motivated by the necessity of developing a next-generation datacenter computing technology that overcomes several important limitations in current solutions. The DVM should provide the basic

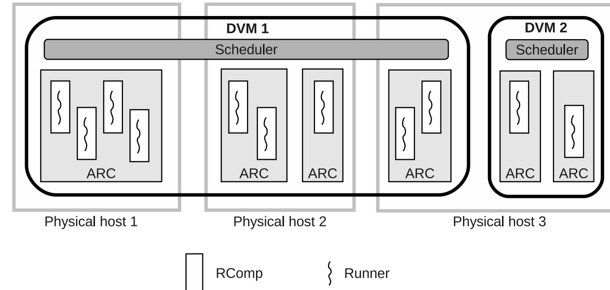

Figure 1. DVM architecture. Two DVMs run on top of three physical hosts in this example.

program execution and parallelization substrate in this technology, and should meet all the goals on *generality*, *efficiency*, *scalability*, *portability*, and *ease-of-programming* (with compilers' help). This leads us to choose the ISA layer to construct a unified computational abstraction of the datacenter platform. To break the *scalability* bottleneck in traditional ISAs and retain the advantages of *generality* and *efficiency*, we design the new ISA, DISA, and build the DVM on this architecture. The design of DISA, the DVM system, and the evaluation of these designs (refer to Section 5) show that the performance of the programs benefits much from the ISA-level abstraction.

We implement the DVM on top of commodity computers and virtual machines, and evaluate it on several platforms with a set of workloads. The results show that DVM can be more than 10 times faster than Hadoop and X10 on certain workloads with moderately iterative logic, and that DVM scales well with near-linear speedup up to at least 256 compute nodes with parallelizable workloads.

The structure of this paper is as follows. We give an overview of DVM in Section 2. Section 3 presents the designs of DISA, memory space, runners, scheduling, etc. Section 4 describes the implementation. We show the evaluation results in Section 5, and discuss related work in Section 6. Section 7 concludes the paper and discusses future work.

2. System overview

The architecture of DVM is shown in Figure 1. A DVM runs on top of one, multiple, or many physical hosts. We abstract a group of computational resources including processor cores and memory to be an *available resource container* (ARC). In its simplest form, an ARC is a physical host. However, it can also materialize as a traditional virtual machine or other container structure, given the container provides an appropriate programming interface and adequate isolation level. Although one ARC is exclusively assigned to one DVM during one period of time, the ARCs in one physical host may belong to one or more DVMs and one DVM's capacity can be dynamically changed by adding or removing ARCs to or from the DVM as needed. Hence, the capacity of a DVM is flexible and scalable—a DVM can share a single physical host with other DVMs, or exclusively include many physical hosts as its ARCs.

2.1 System organization

A DVM accommodates a set of tasks running in a unified address space. Depending on the required resources, multiple DVMs can co-exist on a single physical host, or a single DVM may scale to run on a large number of physical hosts. Although DVMs may have different capacity and the size of a DVM can grow to very large, the DVMs provide the same architectural abstraction of a "single computer" to programs through the Datacenter Instruction Set Architecture (DISA). The programs in DVMs are in DISA

Table 1. DISA instructions

Instruction	Operands	Effect
mov	*D1, M1*	Move [*D1*] to *M1*
add	*D1, D2, M1*	Add [*D1*] and [*D2*]; store the result in *M1*
sub	*D1, D2, M1*	Subtract [*D2*] from [*D1*]; store the result in *M1*
mul	*D1, D2, M1*	Multiply [*D1*] and [*D2*]; store the result in *M1*
div	*D1, D2, M1*	Divide [*D1*] by [*D2*]; store the result in *M1*
and	*D1, D2, M1*	Store the bitwise AND of [*D1*] and [*D2*] in *M1*
or	*D1, D2, M1*	Store the bitwise inclusive OR of [*D1*] and [*D2*] in *M1*
xor	*D1, D2, M1*	Store the bitwise exclusive OR of [*D1*] and [*D2*] in *M1*
br	*D1, D2, M1*	Compare [*D1*] and [*D2*]; jump to *M1* depending on the comparing result
bl	*M1, M2*	Branch and link (procedure call)
newr	*M1, M2, M3, M4*	Create a new runner
exit		Exit and commit or abort

M means this operand is a memory address; *D* means this operand is a memory address or immediate value; [*D*] means the immediate value (when *D* is an immediate value) or the value stored in the memory address (when *D* is a memory address)

instructions and a program running inside a DVM instantiates to be a number of tasks called *runners* executing on many processor cores and computers.

Inside an ARC, many runners, each residing in one *runner compartment* (RComp), can execute in parallel. The RComp is a semi-persistent facility that provides various system functions to runners and the meta-scheduler (discussed later) to facilitate and manage runners' execution. Specifically, the RComp accepts the meta-scheduler's commands to prepare and extend runners' program state and start runners, helps the memory subsystem handle runners' memory accesses, facilitates the execution of instructions such as `newr` (see Table 1), sends scheduling-related requests to the meta-scheduler, and provides I/O services. As these functions are tightly correlated with other subsystems of DVM, we introduce pertinent details of the RComp design along with the treatise of the other system components in Section 3. Suffices it at present to understand that one RComp contains at most one runner at a specific time, and it becomes available for assigning a new runner to after the current incumbent runner exits.

The DVM contains an internal meta-scheduler (or, simply, scheduler) to schedule tasks (runners) executing in RComps. The scheduler is a distributed service running on all constituent hosts of a DVM, with one scheduler master providing coordination and serialization functions. The scheduler dispatches runners to execute in RComps, prepares runners' memory, and handles runners' requests to create new runners.

2.2 Programming in DISA

In this section, we use one example to illustrate how to write a simple program on a DVM using DISA. Details of the DISA architecture will be introduced in Section 3, and a more complex example will be introduced in Section 4.2.

The following DISA code performs a sum operation of two 64-bit integers by a *sum_runner* runner. The starting address of the memory range storing the two integers is in `0x100001000`, and the result's address is in `0x100001008`. The lines starting with "#" are comments and the runners' names are in italic.

```
sum_runner:
add (0x100001000):q, 8(0x100001000), (0x100001008)
                                        # :q indicates 64-b data
mov:z $1:q, 8(0x100001008):q            # set exit code
exit:c                                  # exit and commit
```

With an emphasis on instruction orthogonality, DISA allows the instruction to freely choose memory addressing modes to reference the first and second operands. The second instruction sets the exit code to indicate the result is ready (`:z` means "zero extend"). The last instruction makes the runner exit and commit (`:c`) the changes.

3. Design

We design DISA as a new ISA targeting a large-scale virtual machine running on a plurality of hosts connected through the network inside a datacenter and build the DVM above this architectural abstraction. In this section, we first introduce the design of DISA's instructions then present DISA's memory model in Section 3.2 and the parallelization mechanism in Section 3.3.

3.1 DISA

Given the system goals of the DVM design as listed in Section 1, the goals of the DISA are as follows.

1. DISA should support a memory model and parallelization mechanism scalable to a large number of hosts.
2. DISA programs should efficiently execute on common computing hardware used in datacenters, which are usually made of commodity products. Certain hardware may provide native support to DISA in the future, although we emulate DISA instructions using x86-64 instructions in our current implementation.
3. DISA instructions should be able to express general application logic efficiently.
4. DISA should be a simple instruction set with a small number of instructions so that it is easy to implement and port.
5. DISA should be Turing-complete.

In summary, DISA should be scalable, efficient, general, simple, and Turing-complete. To ensure programmability, simplicity and efficiency, DISA, as shown in Table 1, includes a selected group of frequently used instructions. The `newr` and `exit` instructions in DISA facilitate construction of concurrent programs with a plurality of execution flows.

We show the Turing completeness of DISA by using DISA to emulate a Turing-complete `subleq` machine [39]. The emulator accepts an arbitrary `subleq` program and generates the corresponding output. The DISA implementation is as follows (each `subleq` instruction is encoded as a triple of 64-bit integers representing its three parameters).

```
loop: mov:z (pc):q, addr1:q
      mov:z 8(pc):q, addr2:q
      sub (addr2):q, (addr1), (addr2)
      br:l $0:q, (addr2), next
      mov:z 16(pc):q, addr3:q
      mov (addr3):q, pc
      br:j loop
next: add pc:q, $24, pc
      br:j loop
```

Here `pc`, `addr1`, `addr2` and `addr3` are mnemonic names of the addresses of special variables used by the emulator. `pc` initially points to the start address of the input `subleq` program. The third instruction means "jump to *next* if 0 is less than the value in

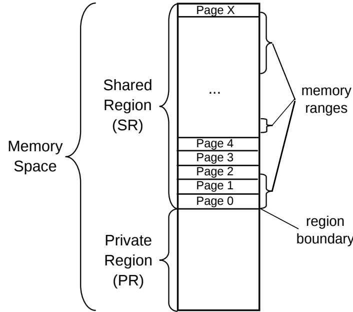

Figure 2. Organization of the memory space

the 64-bit memory range whose address is stored in `addr2`". Being Turing-complete, DISA is capable of implementing any computable application logic.

DISA's architecture does not explicitly contain registers. This design provides a unified operand representation yet does not prevent register-based optimization because the memory model allows some addresses to be affiliated with registers. A DISA instruction may use options to indicate specific operation semantics, such as the `:z` and `:c` in the *sum_runner* example, and the operands can refer to immediate values or memory content using direct, indirect or displacement addressing mode.

3.2 Unified memory space

To facilitate the illusion of programming on a single computer, DISA provides a large, flat, and unified memory space where all runners in a DVM reside. The memory space of a DVM is divided into two regions–the private region (PR) and the shared region (SR). The starting address of the SR is the region boundary (RB) which is just after the address of PR's last byte. The memory layout is shown in Figure 2.

A write in an individual runner's PR does not affect the content stored at the same address in the other runners' PRs. In contrast, the SR is shared among all the runners in the sense that a value written by one runner can be visible to another runner in the same DVM with DISA's memory consistency model controlling when values become visible. A runner can use the PR to store temporary or buffered data because of its very small overhead, and use the SR to store the main application data which may be shared among runners. In our implementation on the x86-64 platform, we set RB to `0x400000000` and an application can use a 46-bit memory address space. Therefore, programs can potentially store around 64TB data in SR and each runner can store several gigabytes of data in its PR, which is sufficient for most of the applications in datacenters.

The consistency model of DISA plays a critical role on the programmability and the scalability. DISA provides a snapshot-based consistency model that relaxes the read/write order on different snapshots. In DISA, a snapshot is a set of memory ranges instantiated with the memory state at a particular point in time. A memory range is a sequence of consecutive bytes starting from a memory address. After a snapshot is created for a runner, later updates to the associated memory ranges by other runners do not affect the state of the snapshot. Hence, when reading data from a snapshot, a runner always receives the value the runner itself has written most recently or the value stored at the time the snapshot is taken. The consistency model of DISA permits concurrent accesses optimistically, and detects write conflicts automatically. With this consistency model, it is possible to implement atomic writes of a group of data, which are commonly used in transactional processing and other reliable systems requiring ACID properties.

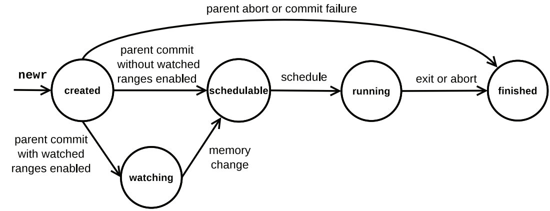

Figure 3. State transition of the runner and watcher

To assist the runners in using the shared and snapshotted SR in the unified memory space, components of the DVM cooperate to manage the snapshots and facilitate the memory accesses by runners. The snapshots are managed by the scheduler for runners and serialized by the memory subsystem. The scheduler creates a snapshot for a runner on its interested memory ranges before scheduling it for execution. The snapshot is associated with the runner throughout its lifetime and the snapshot can be extended only by the scheduler when it is needed. At the end of the runner's execution, the scheduler commits the changes made by the runner to the DVM memory. The commit operation is atomic and the updates are made visible to all the snapshots created thereafter.

The RComps facilitate the memory subsystem to handle the runners' memory accesses. An individual runner's PR is implemented as the local memory ranges on the RComp where the runner resides. The SR is shared among all the runners and distributed across the RComps. The memory subsystem reuses the virtual memory hardware to detect memory accesses in the SR and service remote data following established practices [29, 32].

As locality is crucial for performance in large clusters [49], we design a simple yet general mechanism to coordinate runners to execute on RComps based on the memory range usage history to improve locality for various workloads. The scheduler proactively assigns a runner to the RComp that recently used the SR ranges needed by the runner if such an RComp exists. In the mean time, the users or programmers need not to know or make specific arrangements for the physical memory allocation and distribution.

As updates by one runner only affect the content of the associated snapshot, most of the memory accesses are handled locally at native speed. The order of read/write by different runners is relaxed and concurrent accesses and updates are permitted in this memory model. This makes the system easily scalable with both the number of runners and the data size. In the mean time, the memory subsystem only serializes the committing of memory ranges when runners exit. Hence, the DVM can efficiently manage a large number of snapshots for the runners in the system.

3.3 Tasking and scheduling

We design DVM's parallelization mechanism so that it is easier to develop not only "embarrassingly parallel" programs, but also more sophisticated applications, and the programs can efficiently execute in a datacenter environment. Our goal is to provide a scalable, concurrent, and easy-to-program task execution mechanism with automatic dependence resolution. To achieve the goal, we design a many-runner parallelization mechanism for scalable and efficient concurrent execution, a scheduler to manage the runners and the many-runner execution, and watchers to express and resolve task dependences.

3.3.1 Runner

Runners are the abstraction of the program flows in a DVM. A runner is created by its parent runner and terminates after it exits or aborts. Figure 3 summarizes the state transitions in a runner's life cycle. The parent runner creates a new runner by executing

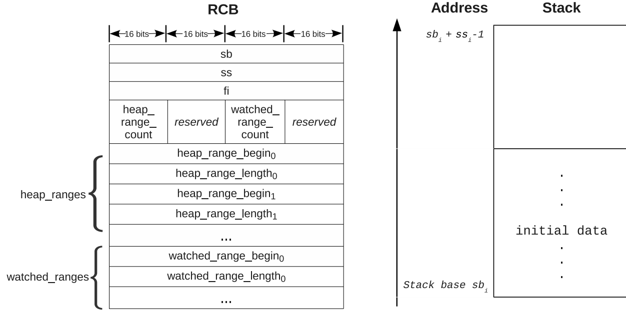

Figure 4. Runner i's initial stack and RCB

the newr instruction. A "created" runner moves to "schedulable" state after the parent runner exits and commits successfully. The runner moves to the "running" state after being scheduled and to the "finished" state after it exits or aborts. The "watching" state applies only to a class of special runners called "watchers", which are to be introduced in Section 3.3.4.

Each runner uses a range of memory as its stack range (stack for short), changes to which are discarded after the runner exits, and multiple ranges of memory as its heap ranges. Runners in a DVM are started by the scheduler. Therefore, the scheduler needs to know certain control information about the runner, such as the location of code and the runner's stack and heap ranges. DVM uses a Runner Control Block (RCB) to store the control information needed by the scheduler to start a runner. Figure 4 shows the content of a runner's RCB as well as the runner's initial stack. The RCB contains the stack base address sb and stack size ss. A 64-bit pointer, fi, specifies the address of the first instruction from which the runner starts to execute. The $heap_ranges$ and $heap_range_count$ fields stores the locations and the number of the heap ranges, respectively. The $watched_ranges$ and $watched_range_count$ fields are used by the watcher mechanism which will be discussed later.

Every runner resides in one RComp. Providing system functions related to the management and operations of runners, the RComp reduces the overhead of dispatching and starting a runner. RComps interact with other parts of the DVM system, and hide the complexity of the system so that the runners can focus on expressing the application logic on the abstraction provided by DISA. Because RComps are reused, a large portion of the runner startup overhead, such as loading modules and setting up memory mapping, is incurred only once in the lifetime of a DVM. To start a runner, the RComp only needs to notify the memory subsystem to set up the runner's snapshot, and the supporting mechanisms, such as the network connections, can be reused. Hence, RComps make it efficient to start runners in a DVM. Section 5 quantitatively studies the overhead of creating and starting runners.

3.3.2 Scheduling

In a complex program, myriads of runners may be created dynamically and exit the system after completing their computation. RComps, on the other hand, represent a semi-persistent facility to support runners' execution. Hence, there can be more runners than RComps in a DVM, and the scheduler assigns runners to RComps, as well as manages and coordinates the runners throughout their life cycles. Specifically, the scheduler is responsible for deciding when runners should start execution, assigning runners to RComps, preparing runners' memory snapshots, handling runners' requests to create new runners, and managing the runners that the DVM does not have resources (RComps) to execute currently. As the child runners created by a parent runner are not schedulable until the parent runner exits and commits successfully, the scheduler also maintains the "premature" runners (the runners in the "created" state) in the DVM. The scheduler also implements the DVM's watcher mechanism (to be discussed in Section 3.3.4).

The scheduler is a distributed service running on all constituent hosts of DVM. Specifically, the scheduler of a DVM consists of two parts: the scheduler master (the master) and the scheduler agent. There is a single scheduler master in a DVM, and there are many scheduler agents residing in RComps. The master and the agents communicate with each other through the datacenter network. The master makes high-level scheduling decisions, such as dispatching runners for execution and assigning them to RComps, and commands the agents to handle the remaining runner management work, such as preparing and updating runners' snapshots. In the other direction, an RComp can send scheduling-related requests to its associated agent and the agent may ask the master for coordination if it is needed.

The scheduler dispatches multiple runners from its pool of schedulable runners according to the scheduling policy, which takes locality along with other factors into consideration, to an RComp when it is free (no runner is running in it), creating a memory snapshot for the runner and notifying the RComp to execute the runner. A runner may create as many child runners as the entire DVM (or datacenter) can handle. When a runner requests to create new runners, the scheduler constructs RCBs for the child runners, and extends the parent runner's memory snapshot so that the parent can access and construct the child runners' stacks. After the parent exits and commits successfully, the scheduler adds the child runners to the runner and watcher pools, and the newly created runners in the runner pool become schedulable.

Although there is only one scheduler master in a DVM, it has not been found that the single master constrains the DVM's scalability. As aforementioned, the master is only responsible for high-level decisions and occasional coordination and arbitration, and is, consequently, very lightweight. Most of the work is handled in parallel by the scheduler agents in the RComps. Hence, the scheduler does not constitute bottleneck in the DVM in practice. Several large-scale systems, such as MapReduce [20], GFS [23], and Dryad [27], follow a similar single-master design pattern.

3.3.3 Many-runner parallel execution

The DVM should be able to accommodate a large number of runners and execute them in parallel to exploit the aggregate computing capacity of many physical hosts. This requires the parallelization mechanism to provide scalable, concurrent and efficient task execution and a flexible programming model to support many-runner creation and execution.

DVM uses the shared-memory model to simplify the program development and hides the underlying distributed system details from programmers. All runners inside one DVM share a common memory region (SR) as discussed in Section 3.2. While a runner may write multiple memory ranges in the SR during its execution, the runner may not want to apply all the changes made by it to the global memory space—for example, some ranges may only provide input or temporary data which are not used by other runners. DISA's memory subsystem enables programmers to control the behaviors of memory ranges so that it can support common cases of memory usage. A runner mainly obtains its input from its initial state comprising its stack and heap ranges, and can also read data from I/O channels during its execution. The runner's initial stack state is constructed by its parent runner, and its output is written into its heap ranges and I/O channels if it is needed. We call this memory usage scheme the SIHO (Stack-In-Heap-Out) model.

Upon completion, a runner specifies whether it wants to commit or abort its changes to the heap ranges. If it commits, the changes

to the heap ranges by the runner are applied to the global memory space, given no conflicts with other runners' memory updates exist. If it aborts, the changes are discarded. The changes to a runner's stack are always discarded upon completion of the runner. As a runner operates on its own snapshot independently of other runners', a large number of runners in a DVM can execute in parallel optimistically assuming there are no conflicts. In the case that conflicts occur, the DVM system automatically detects conflicts, delays execution, and aborts runners, if necessary, to ensure correct memory semantics.

To support many-runner parallel execution on the ISA level, the runner creation and dispatching mechanism must be lightweight and highly efficient. We design an instruction-level runner creation mechanism. To create a new runner and prepare it for execution, the program only needs to specify the fi and the heap ranges for the new runner, issue a "`newr`" instruction, and instantiate the stack.

We use an example to show how a runner creates a new runner and how the DVM handles the runner creation. When a runner, R_i, creates a new runner, R_j, R_i executes the instruction "`newr`" with addresses of R_j's stack range, heap ranges and fi, as operands. $RComp(R_i)$, the RComp in which R_i runs, sends R_j's control information specified by these operands to the scheduler. As R_i may write the input data into R_j's stack and a memory range can be used by R_i only after $RComp(R_i)$ has the range's snapshot, the scheduler creates a snapshot of R_j's stack and merges the snapshot into R_i's current one. After the memory range for R_j's stack is ready, R_i writes the initial application data into R_j's stack. The scheduler constructs and records R_j's RCB along with the RCBs of the other runners created by R_i.

When R_i exits and commits, the scheduler starts the newly created runners by R_i, including R_j, according to the recorded RCBs. Using R_j as an example, we specify the runner start-up procedure in Algorithm 1.

Algorithm 1 Start a new runner R_j

1: $RComp(R_j)$ = the RComp chosen by the scheduler for R_j
2: A = snapshot of R_j's stack range and heap ranges, created by the scheduler according to RCB_j
3: The scheduler sends A and RCB_j to $RComp(R_j)$
4: The scheduler commands $RComp(R_j)$ to start R_j
5: $RComp(R_j)$ sets R_j's local context according to RCB_j
6: R_j starts from fi to execute in $RComp(R_j)$

3.3.4 Task dependency

Task dependency control is a key issue in concurrent program execution. Related to this, synchronization is often used to ensure the correct order of the operations, avoid race conditions, and, with execution order constrained, guarantee that the concurrent execution does not violate dependence relations. For example, X10 provides barriers through the `finish` statement and `clock` operations to conduct synchronization and, consequently, control the execution order [15]. A MapReduce application's map workers' output is the reduce workers' input, which, in turn, implicitly requires dependence be handled in this way. Dryad requires the programmer or compiler to explicitly specify the dependency graph of the programs using a DAG (directed acyclic graph) [27].

As an important goal in the DVM design, the DISA architecture should provide a task dependency representation and resolution mechanism with which task-level dependence can be naturally expressed, concurrent tasks can proceed without using synchronization mechanisms such as locks [11], and sophisticated computation logic can be processed effectively with dynamically scheduled parallelism. The aforementioned approaches have their limitations and constraints: the synchronization mechanisms are not natural or efficient to represent dependence [16, 45]; MapReduce employs a restricted programming model and makes it difficult to express sophisticated application logic [21, 41, 47]; DAG-based frameworks incur non-trivial burden in programming, and automatic DAG generation has only been implemented for certain high-level languages [48]. Departing from existing solutions, we design a watcher mechanism in the DISA architecture to provide a flexible way of declaring dependence and enable the construction of sophisticated application logic. A watcher is a runner that depends on other runners and watchers: a watcher W is not schedulable until other runners or watchers change specific memory ranges that are "watched" by W. We call these ranges W's "watched ranges", and we use the $watched_range_count$ field in the RCB (refer to Figure 4), which stores the number of watched ranges (0 for a regular runner), to distinguish a watcher from a regular runner. The watcher-specific data structures impose almost no space overhead for regular runners as the $watched_ranges$ list, which stores the watched ranges, is variable-size.

Suppose a runner, R, has created the watcher W. When R exits and commits, W is enabled by the scheduler (transition to the "watching" state in Figure 3). Different from a regular runner, W is in the "watching" state, is not schedulable, and stays in the watcher pool. When some other runners or watchers write W's watched ranges and commit the changes to the global memory space, the memory subsystem notifies the scheduler, and the scheduler moves W to the runner pool, making it schedulable.

With the watcher mechanism, the programs focus on and naturally express the true dependence among tasks (runners) and data. Therefore, programmers can easily construct programs for not only embarrassingly parallel but also complex, iterative, or even interactive computation. The DVM system automatically resolves the dependence and efficiently executes the program with parallelization.

3.4 I/O, signals and system services

We design the I/O mechanism to enable runners to read and write data on I/O channels. For simplicity and flexibility, DISA adopts the memory mapped I/O for DVM to operate on I/O channels. This design does not require additional instructions—all DISA's memory access instructions can also be used for I/O. We can use this mechanism to support many kinds and a large number of I/O channels which may be distributed in many RComps. In the evaluation discussed in Section 5, we use I/O channels to input the dataset to the k-means program.

Signals are notifications of the state changes in a DVM. We design the signal mechanism to enable runners to catch the information of its interested events in the system. We design the signal mechanism above the watcher mechanism and the system services (to be discussed below). The signals are represented as changes to specific memory ranges, and runners (watchers) can "watch" these memory ranges to get notified.

Several distributed system services collaborate with the scheduler and RComps to support system functions. For example, the I/O services on RComps manage I/O channels, and the "system state" service broadcasts "system state changing" signals. On top of the compact and efficient core parts of the DVM as described, it is easy to implement various system services to provide a rich set of functions.

4. Implementation and programming notes

We have implemented DVM and DISA on x86-64 hosts. The implementation of the DISA and DVM comprises around 11,510 lines of C and assembly code. On each host, the DVM runs in conjunction with a local Linux OS as the base system, which manages local resources including CPUs and local file systems. The distribution of the code in each part of DVM is shown in Table 2.

Table 2. Implementation of DVM

Sub-system	LOC	Size of binary code (KiB)
Scheduler	2861	59
Memory	3220	72
RComp	715	40
Translator	4714	73

Figure 5. The CCMR research testbed

4.1 Implementation and portability

We have implemented and run DVM on three platforms—a research testbed, an industrial testbed, and the Amazon Elastic Compute Cloud (EC2). Although the three platforms have quite different underlying hardware configurations and slightly different Linux kernels, it proves to be straightforward to port DISA to all these platforms by adding logic to match specific kernel data structures. Application programs do not need to be changed at all on three platforms. Our experience shows that DISA and DVM are highly portable.

We emulate DISA instructions on x86-64 machines in our current implementation using two mechanisms. One is to translate the DISA assembly to x86-64 assembly and compile it to object code. The other is to conduct binary translation to convert x86-64 code on the fly during the execution of a DISA program. Which emulation mode to use in a DVM can be configured at compilation time. In our evaluation discussed in Section 5, we manually encode the microbenchmark programs to run on a DVM. As one direction of future work, we will build compilers for the DISA architecture.

Our main development and evaluation platform is the CCMR research testbed which we can fully control. Shown in Figure 5, the CCMR testbed is constructed for cloud computing research, and is composed of 50 servers with Intel Quad-Core Xeon processors. The testbed has two additional servers providing GPU acceleration, but they are not used in our evaluation.

To verify the performance results in industrial environments, we repeat some experiments on the industrial cluster. The industrial testbed comprises 32 servers, each of which has one Intel Core 2 Duo processor.

To test DVM's scalability in a larger scale, we conduct part of the experiments on Amazon EC2 virtual machines. The results on EC2 also show the portability of DISA in a virtualized and heterogeneous environment.

4.2 How to program

We extend the *sum_runner* example (introduced in Section 2.2) to illustrate how to write a parallel program with data dependence in DISA. The example uses 10,240 runners to compute sums of 20,480 integers in parallel. The execution graph is shown in Figure 6. Each vertex in the graph represents one runner that sums two integers and stores the result in the global memory space.

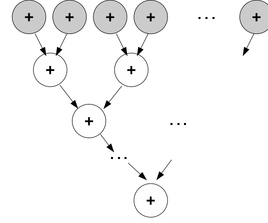

Figure 6. The execution graph of the parallel sum

4.2.1 Parallelism

The following code starts 10,240 *sum_runner* runners to compute 10,240 partial sums. The calculation of the final sum requires additional rounds of sum operations following data dependence, which will be introduced later.

```
main_runner:
# create 10,240 sum_runner runners:
# store the addresses of ranges for stacks and heaps
#   in 0x100001008 and 0x100001010 (omitted)
mov:z $0:q, 0x100001000:q                # i = 0
j_do:
# suppose the addresses of the runner's control information
#  are in 0x1f0000100, 0x1f0000108, 0x1f0000110 and 0x1f0000118
mov:z 0x100001008:q, (0x1f0000100):q     # set stack base
mov:z $0x1000:q, 8(0x1f0000100):q        # set stack size
mov:z $1:q, (0x1f0000108):q              # set heap range count
mov:z 0x100001010:q, 8(0x1f0000108):q    # heap range
mov:z $0x1000:q, 16(0x1f0000108):q       # heap range's length
mov:z $0:q, (0x1f0000110):q              # watched range count
mov:z $sum_runner:q, (0x1f0000118):q     # set code
# create the new runner:
newr 0x1f0000100, 0x1f0000108, 0x1f0000110, 0x1f0000118
# initialize the stack (omitted here)
add $0x1000:uq, 0x100001008, 0x100001008 # update stack base
add $0x1000:uq, 0x100001010, 0x100001010 # update heap range
add $1:uq, 0x100001000, 0x100001000      # i++
br:l 0x100001000:uq, $10240, j_do        # while i < 10240
exit:c                                   # exit and commit
```

In this example, *main_runner* creates 10,240 *sum_runner* runners by executing the `newr` instruction with the new runners' control information as operands. The new runners will be schedulable after *main_runner* exits and commits (the `exit:c` instruction).

From this example, we can see that programs can create new execution flows efficiently at the DISA instruction level, and programmers can easily design and effectively express distributed and parallel programs with the DISA instruction set.

4.2.2 Data dependence

Each of the runners represented as white vertices in Figure 6 depends on the output of two other runners as indicated by the edges in the graph. We show in the following code how to express this data dependence using the "watcher" mechanism.

```
sum_watcher:
# read the heap addresses of the depended runners or watchers and
#   the address to store the result from the stack, and stores
#   them in 0x100001000, 0x100001008, and 0x100001010 (omitted)
# check whether the depended data is ready
br:e $0:uq, 8(0x100001000), create_self
br:e $0:uq, 8(0x100001008), create_self
# sum the two integers as the sum_runner
add (0x100001000):uq, (0x100001008), (0x100001010)
mov:z $1:q, 8(0x100001010):q             # exit code
exit:c
create_self:
```

```
# create sum_watcher with the same control information
#   as itself (omitted)
exit:c

main_runner:
# creates 10,240 sum_runner as in the previous part (omitted)
# creates 10,239 sum_watcher watchers:
mov:z $0:q, 0x100001000:q              # i = 0
j_do2:
# suppose the addresses of the watcher's control information are
#  in 0x1e0000100, 0x1e0000108, 0x1e0000110 and 0x1e0000118,
#  and the watched ranges are calculated according to i and
#  stored in the memory range starting from 0x100001018
# set the stack range, the heap ranges, and the code (omitted)
mov:z 0x100001018:q, 8(0x1e0000110):q    # set watched range 1
mov:z 0x100001020:q, 16(0x1e0000110):q   # set range 1's length
mov:z 0x100001028:q, 24(0x1e0000110):q   # set watched range 2
mov:z 0x100001030:q, 32(0x1e0000110):q   # set range 2's length
# create the new watcher:
newr 0x1e0000100, 0x1e0000108, 0x1e0000110, 0x1e0000118
# initialize the stack (omitted)
# update stack base and heap ranges (omitted)
add $1:uq, 0x100001000, 0x100001000      # i++
br:l 0x100001000:uq, $10239, j_do2       # while i < 10239
exit:c                                   # exit and commit
```

The *main_runner* creates 10,239 *sum_watcher* watchers together with the 10,240 *sum_runner* runners. The *sum_watcher* is activated after the data it depends on are changed by other runners or watchers. As the *sum_runner* (or the *sum_watcher*) writes "1" as the exit code which is also used as the "marker" to indicate that it has committed its heap ranges, the *sum_watcher* can check the exit codes to determine whether both operands for the sum operation are ready. The watcher creates itself again to continue "watching" the data it depends if either of the two depended runners or watchers has not exited and committed.

This example shows that programs can express task dependence easily by watchers on the ISA level without using synchronization mechanisms. The dependences are automatically resolved by the scheduler and watcher-related mechanisms in the DVM.

5. Evaluation

We measure the virtualization and tasking overhead of DVM, compare the performance of DVM with Hadoop and X10, and inspect the scalability of the DVM. We choose Hadoop and X10 for comparison because they are representatives of two mainstream approaches to datacenter computing and their implementations are relatively mature and optimized. Hadoop represents a class of MapReduce-style programming frameworks such as Phoenix [41], Mars [25], CGL-MapReduce [21], and MapReduce online [19], and has been used extensively by many organizations [5]. X10 represents a class of language-level solutions, including Chapel [13] and Fortress [4], targeting "non-uniform cluster computing" environments [15].

For each technical solution in comparison, we apply the highest performance setting among the standard configurations. For example, X10 programs are pre-compiled to native executable programs to give the best performance, and we write the x86-64-based microbenchmark programs in comparison in the assembly language to optimize the baseline to our best. We use measurement programs and two computational workloads to evaluate the DVM on the three platforms presented in Section 4.1, examine the execution time and speedup, and compare DVM's results with other technologies'. Note that the platforms have different performance characteristics. To make the comparison fair, we indicate the platform on which the evaluation is conducted, use the same set of compute nodes and the same network topology and configuration to run comparative experiments on each platform, and repeat the experiments to make sure results are consistent.

Table 3. Microbenchmark results

Benchmark	Solution	Time (second)
Arithmetic and logic operation	Native	7.13
	DVM	7.14
	Xen	7.14
	VMware	7.35
Memory operation	Native	110.60
	DVM	126.41
	Xen	112.43
	VMware	128.37

Table 4. Time for creating and executing a runner

Operation	Operation step	Time (ms)
Create a runner	Create snapshot	0.4
	Create the runner	0.4
	Update RComp	0.5
	Overall	1.3
Execute a runner (empty runner)	Create snapshot	0.4
	Update RComp (NULL)	0.4
	Update RComp	0.5
	Invoke the runner	0.2
	Commit snapshot	0.9
	Overall	2.4

The evaluation results show that DVM is one order of magnitude faster than the other technologies on workloads with moderate sophistication, runs reliable across different environments, and scales easily to hundreds of computer nodes.

5.1 Microbenchmarks

We run microbenchmarks to measure the virtualization overhead of DVM and compare the results with traditional virtual machine monitors. We also measure the overhead of creating and executing runners in a DVM.

5.1.1 Virtualization overhead

We measure the arithmetic and logic operation performance to assess the overhead of CPU virtualization. A microbenchmark program is created to execute totally 2^{34} *add* operations, and implemented in x86-64 assembly and DISA. We execute the microbenchmark programs in Xen, VMware Player, native Linux, and a DVM on the same physical host. To achieve the best performance, the x86-64 assembly implementation heavily uses registers. The DISA implementation is assembled into DISA binary code, and the binary translator of DVM translates DISA instructions into x86-64 instructions during the execution.

Memory is another important subsystem in the DVM, and we shall verify that the large-scale, distributed and shared memory system does not incur dramatic overhead on one machine. We measure the memory operation performance with another microbenchmark program that writes 2^{29} 64-bit integers (4 GB in total) to the memory for 256 times and then read them for 256 times. We implement the microbenchmark in x86-64 assembly and DISA and execute the microbenchmark on the same physical host.

Table 3 shows the results of the microbenchmarks. From the result, we can see that the virtualization overhead for arithmetic, logic and memory operation of DVM is 0.1–15% over native execution on the physical host, which is comparable to traditional VMMs. In particular, DVM exhibits even higher performance than VMware on one physical host. For the memory microbenchmark, the time on DVM includes 7.47 seconds for creating and committing the memory snapshot for the runner.

5.1.2 Runner overhead

Task creation, scheduling, and termination are important and frequent operations in a task execution engine. To examine DVM's performance in this aspect, we measure the overhead of creating and executing a runner. We use one runner to create 1000 new empty runners and execute these runners. To measure the overhead accurately, we force the tasking operation to be serial by conducting the experiment using one RComp on one working node so that we can obtain the average execution time of each operation.

Table 4 shows the overhead of creating and executing a runner and the time used in each step—create snapshot for the runner, initialize the RComp's memory (update RComp with a NULL snapshot), prepare the memory (update RComp), invoke the runner and commit the snapshot after the runner exits. From the results, we can see that the overhead of creating and executing a runner in a DVM is only several milliseconds. The small overhead enables DVM to handle thousands of runners efficiently, and gives programmers the flexibility to use a large number of runners as they want.

5.2 Workloads

To evaluate DVM's performance of parallel execution, we design and run the prime-checker which uses 1000 runners to check the primality of 1,000,000 numbers. To show DVM's performance on widely used algorithms that are less easy to parallel, we implement the k-means clustering algorithm [35] which iteratively clusters a large number of 4-dimensional data points into 1000 groups and reflects known problem configurations in related work [28, 31, 50].

Prime-checker We use the prime-checker to evaluate DVM's scalability and portability. A runner (the main runner) creates 1000 runners and each runner checks 1000 large numbers. To show the ease of programming with shared memory, we design the program in the way that all the runners use the same global parameters by sharing a memory range so that each runner can calculate its set of numbers to be checked. As the prime-checker is highly parallelizable, we use this arithmetic application to evaluate the scalability of the DVM—whether DVM can achieve near-linear speedup as we add more working nodes.

K-means clustering K-means clustering is an algorithm used for partitioning a data set into `k` clusters iteratively [35], which is commonly used for data mining. Various software and libraries implement the k-means algorithm [6, 36, 44]. Researchers also use k-means algorithm to evaluate MapReduce for machine learning and data intensive analytics [18, 21]. The k-means process starts with `k` initial clusters and iteratively refines the clusters by reassigning points to the closest cluster and updating the clusters' means. We implement the k-means algorithm mainly following the widely used ones from Mahout [6] and X10's distribution. To achieve reasonably optimized performance results, we optimize the implementations for all three technologies, Hadoop, X10, and DVM, with the methods used in related work [18, 51]. Our implementations are downloadable from `http://baijia.info/dvm/kmeans.tar`. In addition to the same implementation and optimization method, we use the same data set and identical configurations for the three technologies, including the number of iterations, the initial cluster centroids and the value of `k`.

5.3 Performance comparison

We run the performance evaluation programs with the same dataset on different numbers of working nodes on the research testbed to compare the performance and scalability of the three technologies. To verify the results, we run all the tests on the industrial testbed and compare the results with those on the research testbed. We also increase the size of the input data and test with a fixed number of working nodes on the research testbed to evaluate DVM's scalability with data size.

Table 5. Execution time for prime-checker

Number of nodes	Hadoop (second)	DVM (second)
1	16661	15795
2	8352	7885
4	4169	3950
8	2090	1975
16	1112	986

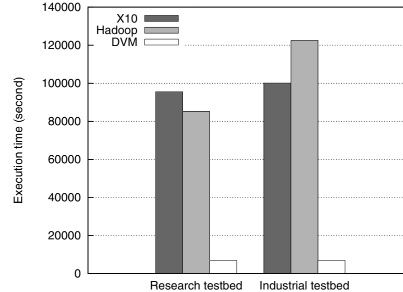

Figure 7. Execution time for k-means on 1 node

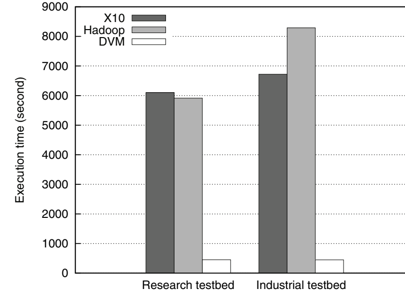

Figure 8. Execution time for k-means on 16 node

Table 5 shows the time of prime-checker on Hadoop and DVM. We conduct the test on the research testbed and scale the number of working nodes to 16 physical hosts. The results show that both Hadoop and DVM scale linearly as prime-check is easy to parallelize. For simple ALU-intensive workloads, both DVM and Hadoop are efficient, but DVM provides slightly higher performance in spite of DVM's advanced instruction set features and memory model.

Figure 7 presents the execution time of k-means on Hadoop, X10 and DVM with one working node on the research testbed and industrial testbed. The execution time on Hadoop and X10 is 11 times more than that on DVM in both research and industrial environments. As shown in this evaluation, DVM quickly exhibits superiority in efficiency as the complexity of application grows. Figure 8 shows the results on 16 working nodes and we can see that DVM is at least 13 times faster than Hadoop, an indication of good scalability for DVM.

Figure 7 and Figure 8 present that DVM exhibits tremendous speed gains over Hadoop and X10, although Table 5 shows that DVM only provides slightly better performance than Hadoop with simple ALU-intensive workloads. This observation, in fact, illustrates the advantages of the DVM technology. When the computation is composed of one or two stages of easily parallelizable processing, the MapReduce/Hadoop frameworks can already accomplish algorithmically optimal performance. The DVM can still

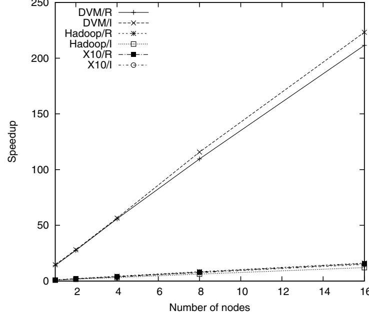

Figure 9. Speedups of k-means as the number of working nodes grows

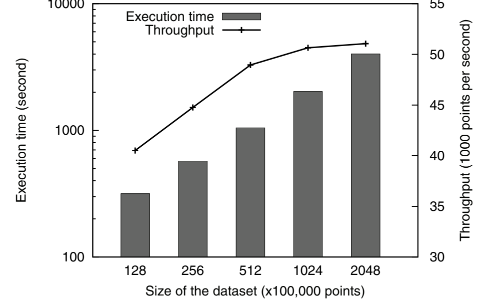

Figure 10. Execution time and throughput of k-means as the size of dataset grows

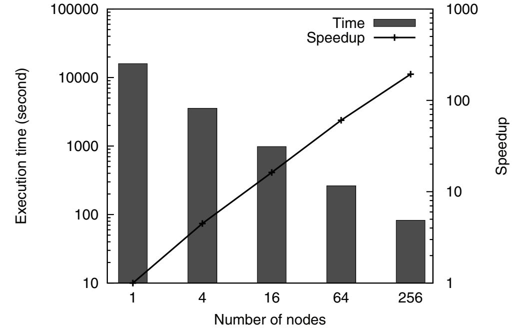

Figure 11. Speedup and execution time of prime-checker as the number of working nodes grows

provide slightly higher performance due to its efficient instruction-level abstraction. Moreover, the DVM quickly outperforms existing technologies when the computation becomes more complex—i.e., containing iterations, more dependence, or non-trivial logic flows, which are common cases in programs. The system design of DVM aims to provide strong support for general programs, and the speed gains of DVM shown in Figure 7 and Figure 8 reflect DISA's generality, efficiency and scalability.

Figure 7 and Figure 8 also show that the Hadoop program on the industrial testbed is slower than on the research testbed. Hadoop transfers data through HDFS [20], the underlying data persistence layer which stores data in working nodes' hard drives, and the working nodes in the research testbed are equipped with higher-end hard drives than the industrial testbed. We believe that the disk I/O contributes to the difference of the program execution time between the research testbed and the industrial testbed. In a cluster with high-speed network (such as 10Gbps Ethernet), which we believe will become more popular in datacenters in the future, the disk I/O speed has certainly an impact on the system performance. On the other hand, DVM uses shared memory for data communication which does not rely on hard drives. From the results, we can see that the performance data of DVM in both environments are close to each other.

It may appear that the DVM's memory-based data processing is the main reason for its superiority in performance. This is, in fact, an over-simplified view of the technical design space. X10 also handles program data through memory with PGAS (Partitioned Global Address Space) [15]. As shown in Figure 7 and Figure 8, however, the DVM is one order of magnitude faster than both Hadoop and X10. In fact, disk I/O does not necessarily outweight other factors, such as network bandwidth and CPU capacity, in a datacenter computing environment. In our evaluation, we have implemented the same algorithm in all the three technologies and applied salient optimizations for each of them—e.g., compiling X10 programs to native x86-64 instructions—to ensure the workloads are comparable with the three implementations. It is the holistic design and the simple yet effective mechanisms in the DVM that lead to the significant speedup over existing technologies, and it is the ISA-level abstraction that enables such a design and the mechanisms. As the result, DVM performs at least as well as other technologies on all workloads, and delivers much better performance than others for complex programs.

Figure 9 shows the speedups for three technologies on the research testbed ("/R" in the figure) and industrial testbed ("/I" in the figure) as we scale the number of working nodes. All speedups are calculated with respect to the X10 performance on one working node for each workload.

The speedup of the k-means computation on DVM is near-linear in the number of compute nodes used, which highlights the good scalability of the design and implementation of DVM. We can also observe that the overhead of DVM is small, taking into consideration the near-linear speedup and the fact that the execution time on DVM in the 16-node experiment is very short compared to the time on Hadoop or X10. The DISA program sequentially reads the input from I/O in our current implementation, and the I/O time is also included in the overall execution time. The k-means computation on DVM should achieve a larger speedup if DVM implemented the parallel I/O which is one direction of future work.

Figure 7, Figure 8, and Figure 9 also show that the execution time and speedups are consistent on the industrial testbed and the research testbed, which verifies the applicability of the DVM technology in industrial environments and reflects DVM's portability.

5.4 Scalability

We run k-means on DVM with 50 working nodes on the research testbed to evaluate DVM's scalability with the data size. Figure 10 presents the execution time and throughput. We scale the dataset from 12,800,000 points to 204,800,000 points (16 times). The throughput is calculated by dividing the number of points by the execution time. From the result, we can see that, on the same number of working nodes, the execution time grows slower than linear growth with the data size while the throughput increases. Hence, DVM scales well with the data size.

To evaluate DVM's scalability and also test DVM's performance in a public cloud, we test the prime-checker on a DVM running on Amazon EC2 with up to 256 virtual machine instances. Figure 11 shows the execution time and speedup with respect to the performance on one instance. The results show that the DVM scales very well to 256 working nodes—the speedup is near-linear as we increase the number of instances. Our ability to perform tests with larger-scale and more sophisticated workloads is limited by the concurrent instance cap stipulated by EC2. In our future work,

we will communicate with Amazon to explore opportunities of larger-scale tests. In the mean time, the current result on 256 instances already shows that, although prime-checker is a relatively easy-to-parallelize program, the DISA architecture and the DVM design easily scale to hundreds of compute nodes without showing signs of slowdown or obvious system bottlenecks. The experiment on EC2 also proves the good portability of DVM in a virtualized environment, and that the DVM can be used independently of or in combination with traditional virtual machine monitors.

6. Related work

Many systems, programming frameworks, and languages are proposed and designed to exploit the computing power of constellations of compute servers inside today's gigantic datacenters and help the programmers design parallel programs easily and efficiently. Dean et al. have created the MapReduce programming model to process and generate large data sets inside Google's datacenter environment [20]. MapReduce exploits a restricted programming model so that the execution engine can automatically run the programs in parallel and provide fault-tolerance. Dryad takes a more general approach by allowing an application to specify an arbitrary "communication DAG (directed acyclic graph)" [27]. While these frameworks are successful in large data processing, the restricted programming model is not general enough to cover many important application domains and makes it difficult to design sophisticated and time-sensitive applications [21, 34, 41, 47]. Different from the previous solutions, DVM allows programmers to easily design general-purpose applications running on a large number of compute nodes by providing a more flexible yet highly scalable programming model.

Although programming frameworks provide simple programming models and interfaces for designing and developing large distributed applications, a new language may help programmers write clearer, more compact and more expressive programs. High-level languages, such as Sawzall and DryadLINQ, which are implemented on top of programming frameworks (MapReduce and Dryad), make the data-processing programs more easier to design [40, 48]. However, these languages also suffer from the limitation of the underlying application frameworks although they allow programmers to design parallel programs using a higher-level abstraction. Charles et al. design X10 to write parallel programs in non-uniform cluster computing system, taking both performance and productivity as its goals [15]. The language-level approach gives the programmers more precise control of the semantics of parallelization and synchronization. DVM provides a lower level abstraction by introducing a new ISA—DISA. On top of DISA, we may implement various programming languages easily using DVM's the parallelization and concurrency control mechanism along with the memory model.

Different from the frameworks and languages that make the data communication transparent to programmers, message passing-based technologies [22] require that the programmer handle the communication explicitly [33]. This can simplify the system design and improve scalability, but often imposes extra burden on programmers. In contrast, the distributed shared memory (DSM) keeps the data transportation among computers transparent to programmers [37]. The DSM systems, such as Ivy [32] and Treadmark [29], combine the advantages of "shared-memory systems" and "distributed-memory systems" to provide a simple programming model by hiding the communication mechanisms [37], but incur a cost in maintaining memory coherence across multiple computers. In order to reduce overhead and enhance scalability, DVM provides snapshotted memory, and the memory consistency model of DISA permits concurrent accesses optimistically. In contrast to the traditional ways of providing DSM in middleware which provide a limited illusion of shared memory, vNUMA uses virtualization technology to build shared-memory multiprocess (SMM) system and provides a single system image (SSI) on top of workstations connected through an Ethernet network that provides "sender-oblivious total-order broadcast" [14]. This also differs from widely used virtualization systems, such as Xen and VMware on the x86 ISA, which divide the resources of a single physical host into multiple virtual machines [8, 46]. Both vNUMA and the emerging "inverse virtualization" [26] aim to merge a number of compute nodes to be one larger machine. Different from vNUMA, which is designed to be used on a small cluster [14], DVM virtualizes the servers at the ISA level and allows programs to scale up to thousands of logic flows across a large number of nodes.

7. Conclusions and future work

Cloud computing is an emerging and important computing paradigm backed by datacenters. However, developing applications running in datacenters is challenging. We design and implemented DVM as an approach to building datacenter-size virtual machines. Giving programmers an illusion of a "big machine", we design the DISA as the programming interface and abstraction of DVM. We implement and evaluate DVM on research, industrial and public clusters, and show that DVM is one order of magnitude faster than Hadoop and X10 for moderately complex applications, and can scale to hundreds of computers.

Beyond the current implementation of DVM, we see a number of interesting future research directions and several of them are as follows. First, we believe that the DISA architecture can scale to larger-scale clusters, and will actively explore opportunities for evaluations of DVM on 1000 or more compute nodes. Second, we shall certainly not require all the programmers write application in DISA. A compiler that "understands" DISA well to generate efficient code is needed on DVM. Third, we may scale a DVM to run across datacenters, so that DVM can use more computing resources and recover from disasters that may possibly render a datacenter inoperable.

8. Acknowledgments

This work was supported in part by the Huawei Technologies research grant HUAW17-15G00510/11PN and HKUST research grants REC09/10.EG06 and DAG11EG04G. We thank Yanling Zheng, Mengmeng Cheng, Chengqi Song and Yanqun Zhang for their help in various aspects of this project, and the Amazon AWS research grant for the support in the EC2 based evaluation. Finally, we are thankful to our shepherd, Ada Gavrilovska, and the reviewers for their reviewing effort and valuable feedback.

References

[1] Amazon Elastic Compute Cloud – EC2. *http://aws.amazon.com/ec2/*. [last access: 11/2, 2011].

[2] Windows Azure. *http://www.microsoft.com/windowsazure/*. [last access: 11/2, 2011].

[3] Rackspace. *http://www.rackspace.com/*. [last access: 11/2, 2011].

[4] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. Maessen, S. Ryu, G. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund, et al. The Fortress language specification. *https://labs.oracle.com/projects/plrg/fortress.pdf*, 2008. [last access: 11/2, 2011].

[5] Apache Hadoop. Hadoop Users List. *http://wiki.apache.org/hadoop/PoweredBy*. [last access: 11/2, 2011].

[6] Apache Mahout. Mahout machine learning libraries. *http://mahout.apache.org/*. [last access: 11/2, 2011].

[7] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above

the clouds: A Berkeley view of cloud computing. *UC Berkeley Technical Report UCB/EECS-2009-28*, February 2009.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM symposium on Operating Systems Principles*, pages 164–177, 2003.

[9] L. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 4(1):1–108, 2009.

[10] L. Barroso, J. Dean, and U. Hoelzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[11] K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.

[12] R. Buyya, T. Cortes, and H. Jin. Single system image. *Intl. Journal of High Performance Computing Applications*, 15(2):124, 2001.

[13] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291, 2007.

[14] M. Chapman and G. Heiser. vNUMA: A virtual shared-memory multiprocessor. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, 2009.

[15] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *ACM SIGPLAN Notices*, volume 40, pages 519–538, 2005.

[16] D.-K. Chen, H.-M. Su, and P.-C. Yew. The impact of synchronization and granularity on parallel systems. In *Proceedings of the 17th annual intl. symposium on Computer Architecture*, pages 239–248, 1990.

[17] Y. Chen, D. Pavlov, and J. F. Canny. Large-scale behavioral targeting. In *Proc. of the 15th ACM SIGKDD intl conf. on Knowledge discovery and data mining*, pages 209–218, 2009.

[18] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-Reduce for machine learning on multicore. In *Proc. of NIPS'07*, pages 281–288, 2007.

[19] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *Proceedings of the 7th USENIX conf. on networked systems design and implementation*, pages 21–21, 2010.

[20] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *the 6th Conference on Symposium on Operating Systems Design & Implementation*, volume 6, pages 137–150, 2004.

[21] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for data intensive scientific analysis. In *Fourth IEEE International Conference on eScience*, pages 277–284, 2008.

[22] M. P. I. Forum. MPI: A message-passing interface standard. `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`, 2009. [last access: 11/2, 2011].

[23] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. of the 9th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 29–43, 2003.

[24] B. Hayes. Cloud computing. *Communications of the ACM*, 51(7): 9–11, 2008.

[25] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th international conference on parallel architectures and compilation techniques*, pages 260–269, 2008.

[26] B. Hedlund. Inverse virtualization for internet scale applications. `http://bradhedlund.com/2011/03/16/inverse-virtualization-for-internet-scale-applications/`. [last access: 11/2, 2011].

[27] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.

[28] H. Jégou, M. Douze, and C. Schmid. Improving bag-of-features for large scale image search. *International Journal of Computer Vision*, 87(3):316–336, 2010.

[29] P. Keleher, A. Cox, S. Dwarkadas, and W. Treadmarks. Distributed shared memory on standard workstations and operating systems. In *Proc. 1994 Winter Usenix Conference*, pages 115–131, 1994.

[30] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[31] D. Lee, S. Baek, and K. Sung. Modified k-means algorithm for vector quantizer design. *Signal Processing Letters, IEEE*, 4(1):2–4, 1997.

[32] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, 1989.

[33] H. Lu, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. In *Proc. of the IEEE/ACM Supercomputing 95 Conf.*, page 37, 1995.

[34] Z. Ma and L. Gu. The limitation of MapReduce: A probing case and a lightweight solution. In *Proc. of the 1st Intl. Conf. on Cloud Computing, GRIDs, and Virtualization*, pages 68–73, 2010.

[35] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, page 14, 1967.

[36] MathWorks. Inc. Matlab. `http://www.mathworks.com/products/matlab/`. [last access: 11/2, 2011].

[37] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[38] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. In *Proc. of the 9th IEEE/ACM Intl. Symposium on Cluster Computing and the Grid*, pages 124–131, 2009.

[39] P. J. Nurnberg, U. K. Wiil, and D. L. Hicks. A grand unified theory for structural computing. *Metainformatics*, 3002:1–16, 2004.

[40] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[41] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proc. of the 2007 IEEE 13th Intl Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[42] Salesforce.com. `http://www.salesforce.com`. [last access: 11/2, 2011].

[43] M. C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25:1363–1369, 2009.

[44] The R Project. The R Language. `http://www.r-project.org/`. [last access: 11/2, 2011].

[45] C. Tseng. Compiler optimizations for eliminating barrier synchronization. In *ACM SIGPLAN Notices*, volume 30, pages 144–155, 1995.

[46] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[47] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: simplified relational data processing on large clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040, 2007.

[48] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *the 8th Conference on Symposium on Operating Systems Design & Implementation*, pages 1–14, 2008.

[49] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys '10: Proceedings of the 5th European conference on computer systems*, pages 265–278, 2010.

[50] R. Zhang and A. Rudnicky. A large scale clustering scheme for kernel k-means. *Pattern Recognition*, 4:40289, 2002.

[51] W. Zhao, H. Ma, and Q. He. Parallel k-means clustering based on mapreduce. In *roceedings of the First International Conference on Cloud Computiong (CloudCom)*, pages 674–679, 2009.