DVM: A Big Virtual Machine for Cloud Computing

Zhiqiang Ma, Zhonghua Sheng, Lin Gu *Member, IEEE*

Abstract—As cloud-based computation grows to be an increasingly important paradigm, 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 four platforms, and our evaluation shows that DVM has excellent performance and scalability. 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 160 compute nodes in the TH-1/GZ supercomputer, the DVM delivers a 12.99x speedup over the computation on 10 compute nodes. The implementation of DVM also allows it to run above traditional VMMs, and we verify that DVM shows linear speedup on a parallelizable workload on 256 large EC2 instances.

Index Terms—Distributed Systems, Concurrent Programming, Datacenter, Virtualization, Cloud Computing

1 Introduction

Virtualization is a fundamental component in cloud technology. Particularly, ISA-level virtual machine monitors (VMM) are used by major cloud providers for packaging resources, enforcing isolation [1], [2], and providing underlying support for higher-level language constructs [3]. However, the traditional VMMs are typically designed to multiplex a single physical host to be several virtual machine instances [4], [5], [6]. Though management or single system image (SSI) services can coordinate multiple physical or virtual hosts [6], [7], [8], 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 instruction set to a cluster of machines [10], but finds it necessary to simplify the memory semantics and encounters 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 [11], [12] and productivity-oriented parallel programming languages [13], [14], [15], [16] have been widely used in cluster-wide computation. MapReduce and its open-source variant, Hadoop, are perhaps the most widely-used framework in this category [17]. 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 support sophisticated application logic [17], [18], [19], [20]. The language-level solutions, such as X10 [14], Chapel [21] and Fortress [13], 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 architecture 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 programmers the view of a

• *Z. Ma, Z. Sheng and L. Gu are with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology. E-mail: {zma,szh,lingu}@cse.ust.hk*

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 cluster-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 [6], Xen [4], vNUMA [10]) 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 adoption of new ISAs. Fortunately, the cloud computing paradigm presents opportunities for innovations at this basic system level—first, a 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".

As an approach to supporting computing in datacenters, the design of DVM has the following goals.

- **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. 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

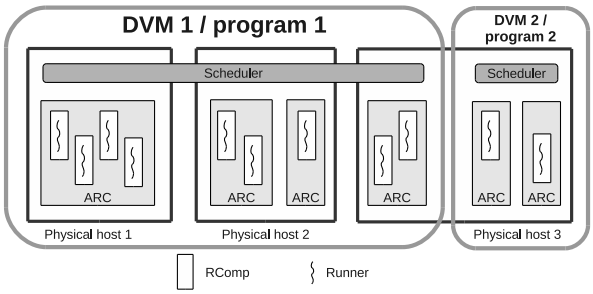


Fig. 1: Organization of 2 DVMs on 3 physical hosts.

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 design. 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 potentially 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 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.

We implement and evaluate DVM on several platforms. The evaluation results (refer to Section 4) show that the performance of the programs benefit much from the ISA-level abstraction—DVM can be more than 10 times faster than Hadoop and X10 on certain workloads with moderately iterative logic, and scale up with near-linear speedup to at least 256 compute nodes on 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 and parallelization mechanisms. Section 4 describes the implementation and the evaluation results. We discuss related work in Section 5 and conclude the paper in Section 6.

2 System Overview

The architecture of DVM is illustrated in Fig. 1. Many DVMs can run in one physical host as the traditional VMMs to multiplex the resources of the host, and a DVM can run on top of multiple or many physical hosts to form a big virtual machine in a datacenter.

2.1 System organization

We abstract a group of computational resources including processor cores and memory to be an *available resource container* (ARC). In the simplest form, an ARC is a physical computer. 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. Hence, the capacity of a DVM is flexible and scalable—a DVM can share a single physical host with other DVMs, or include many physical hosts as its ARCs.

A DVM accommodates a set of tasks running in a unified address space. Although DVMs may have different capacity and the size of a DVM can grow very large depending on the available resources, the DVMs provide the same architectural abstraction of a "single computer" to programs through the DISA.

The programs in DVMs are in DISA instructions and a program running inside a DVM instantiates to be a number of tasks called *runners* executing on many processor cores in many ARCs. 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 a meta-scheduler (discussed later) to facilitate and manage runners' execution. Specifically, the RComp a) accepts the meta-scheduler's commands to prepare and extend runners' program state and start runners, b) helps the memory subsystem handle runners' memory accesses, c) facilitates the execution of instructions such as `newr` (see Table 1), d) sends scheduling-related requests to the meta-scheduler, and e) 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. Suffice it at present to know that one RComp contains at most one runner at a specific time, and it becomes available for accommodating another runner after the current incumbent runner exits.

The DVM contains an internal meta-scheduler (scheduler) to schedule 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.

2.2 Programming in DISA

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 is introduced in [22].

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 `0x300001000`, and `0x300001008` contains the address where the results should be stored. The text after ";" in a line is a comment, and the runner's name is in italic.

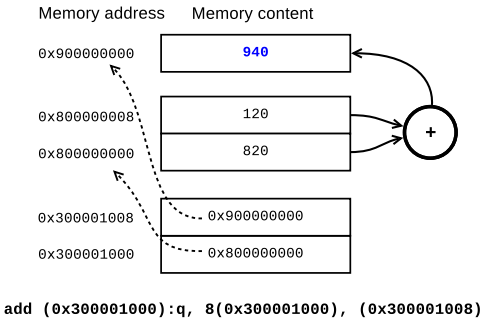


Fig. 2: Execution of an `add` instruction.

```
1  sum_runner:
2  add (0x300001000):q, 8(0x300001000), (0x300001008)
3                                              ; :q indicates 64-b data
4  mov:z $1:q, 8(0x300001008):q                ; set exit code
5  exit:c                                      ; exit and commit
```

The code above adds two 64-bit integers with all operands stored in memory. Fig. 2 shows the execution of the `sum` operation. Suppose the memory location `0x300001000` and `0x300001008` store values `0x800000000` and `0x900000000`, respectively. The `add` instruction at line 2 obtains the operands at `0x800000000` and `0x800000008`, computes the sum, and stores the sum in `0x900000000`. The `mov:z` instruction at line 4 sets the exit code for the runner with zero-extension at address `8(0x300001008)` (`0x900000008` after dereferencing). Finally, the `exit:c` instruction at line 5 makes the runner exit and commit its changes to the memory. After being committed, the changes become visible to other runners.

The example illustrates several important design choices of DISA. First, it uses memory addressing, not combined register-memory addressing, for all operands. With an emphasis on instruction orthogonality, DISA allows an instruction to freely choose memory addressing modes to reference the operands. Second, DISA provides a low-level abstraction which is close to hardware and the semantics of most of the instructions are common in other widely used ISAs. With this design, DISA retains traditional ISAs' advantages, such as generality and efficiency, and it is easy to map DISA to hardware or another common ISA at low cost. Finally, the commit operation, which makes a runner's changes to the memory visible to other runners, occurs at the end of a runner's execution. This arrangement enforces a clearly defined runner lifecycle and its semantics concerning the program state of the DVM.

3 Design

We design DISA as a new ISA targeting a large-scale virtual machine running on a plurality of networked hosts and build the DVM above this architectural abstraction. In this section, we first introduce the design

TABLE 1: DISA instructions

Instr.	Operands	Effect
mov	D_1, M_1	Move D_1 to M_1
add	D_1, D_2, M_1	Add D_1 and D_2; store the result in M_1
sub	D_1, D_2, M_1	Subtract D_2 from D_1; store the result in M_1
mul	D_1, D_2, M_1	Multiply D_1 and D_2; store the result in M_1
div	D_1, D_2, M_1	Divide D_1 by D_2; store the result in M_1
and	D_1, D_2, M_1	Store the bitwise AND of D_1 and D_2 in M_1
or	D_1, D_2, M_1	Store the bitwise OR of D_1 and D_2 in M_1
xor	D_1, D_2, M_1	Store the bitwise exclusive OR of D_1 and D_2 in M_1
br	D_1, D_2, M_1	Compare D_1 and D_2; branch to M_1 if a specific condition is met
bl	M_1, M_2	Branch and link (procedure call)
newr	M_1, M_2, M_3, M_4	Create a new runner
exit		Exit and commit or abort

M_n specifies an operand in memory; D_n specifies an operand in memory or an immediate operand.

of DISA's instructions, and then present DISA's memory model in Section 3.2 and the parallelization mechanism in Section 3.3.

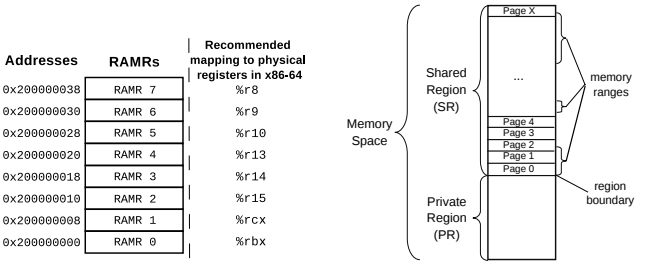
3.1 DISA

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

- DISA should support a memory model and parallelization mechanism scalable to a large number of hosts.
- DISA programs should efficiently execute on common computing hardware used in datacenters, which are usually made of commodity components. Certain hardware may provide native support to DISA in the future, although we emulate DISA instructions on the x86-64 architecture in our current implementation.
- DISA instructions should be able to express general application logic efficiently.
- DISA should be a simple instruction set with a small number of instructions so that it is easy to implement and port.
- 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. In addition, the `newr` and `exit` instructions facilitate construction of concurrent programs with many execution flows. Our prior work shows the Turing completeness by using DISA to emulate a Turing-complete `subleq` machine [22].

A DISA instruction may use options to indicate specific operation semantics, such as the options `:z` and `:c` in the *`sum_runner`* example in Section 2.2, and the operands can reference immediate values or memory contents using direct, indirect or displacement addressing modes. Although not explicitly providing registers, the unified operand representation permits register-based optimization because the memory model allows some memory ranges in DR (*Direct-addressing Region*, discussed in Section 3.2), which can only be referenced with direct addressing, to be affiliated with registers. We



(a) Mapping of RAMRs to physical registers in x86-64

(b) Organization of the memory space

Fig. 3: Organization of the memory space and RAMRs

name these ranges *Register-Affiliated Memory Ranges* or RAMR for short.

Fig. 3(a) shows the RAMR to physical register mapping in our implementation on x86-64. DISA currently supports 8 registers named RAMR 0 to 7. The programmers or compilers can use these RAMRs to store most frequently accessed data in the program to improve the performance. In our evaluation discussed in Section 4, we show that using RAMRs can speed up the *loop* microbenchmark by more than 5 times.

At first glance the RAMR design seems to make DISA less portable. However, the programs using RAMRs need not to be changed for DVMs on different platforms. On a platform that has fewer physical registers in the processors than the RAMRs, the RAMRs simply revert to direct-addressed memory ranges. Hence, a DVM can speed up the programs with physical registers as much as possible while ensuring the portability and correctness.

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 layout is shown in Fig. 3(b). 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. Inside PR, there is a special region, *direct-addressing region* (DR), in which memory ranges

can only be referenced in the direct addressing mode. The DR allows the program to access system resources, such as RAMR (discussed in Section 3.1) and I/O channels (to be discussed in Section 3.4), with the unified operand representation using memory addressing.

A write in an individual runner's PR does not affect the content stored at the same address in 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 the SR and each runner can store several gigabytes of data in its PR, which is sufficient for most of the applications in datacenters.

Snapshot-based consistency. The consistency model of a shared memory system specifies how the system appears to programmers, placing restrictions on the values that can be returned by a read, and is critical for the system's performance [23]. In DISA, a runner's memory operations have a transaction-like semantics and completes as an atomic unit. We use the snapshot-based memory consistency model, in which every runner sees a consistent view of the memory space of a specific instantiation time, and the system imposes a sequential ordering of runners' commits. DISA's consistency model allows a runner to read data in its snapshot and proceed to commit successfully without being disrupted by concurrent updates and commits from another runner, providing the same properties as standard snapshot isolation [24], which is adopted by many relational databases.

DISA's snapshot-based consistency model *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 an address in a snapshot, a runner always receives the value the runner itself has written most recently or the value stored at the address at the time the snapshot is created. 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. This model indeed introduces slight constraints on the organization of programs: only one of many runners that concurrently write to the same location can commit successfully and the other runners need to give up or retry. However, it assures programs written this way are much easier to understand and reason about their correctness, as well as improves the system's scalability.

Managing snapshot metadata. The consistency model of DISA provides programs a clearly-defined memory semantics and enables scalable parallel processing. However, it is challenging to *efficiently* manage a large number of snapshots in the memory space. The snapshot's semantics, which ensures that any later changes of the memory content should not affect the existing snapshots, requires that the memory subsystem maintain more than one version of the memory content. To efficiently support the snapshot management, the memory subsystem organizes the SR as a sequence of fixed-size pages and each page has two versions—a *current version* and a *non-current version*. The current version is the last committed version, and a snapshot consists of a number of pages of their current versions at the time the snapshot is created. The non-current version of a page is not included in newly created snapshots. A page's previous version before the current one, which may be still used by some runners, can be tracked with the non-current version by the memory subsystem.

With the two-version design, the memory subsystem need to only maintain a small amount of metadata for the pages, and can efficiently handle a large number of snapshots. Meanwhile, the current and non-current version design is friendly to the read-only sharing of data—many runners can read shared data with snapshot isolation and complete their execution given they write to disjoint locations. The two-version design indeed implies a slight limitation that some runners may need to wait for available page versions. Typically, there are many runners schedulable in a DVM and the waits usually occur to a small fraction of them. Hence, the limitation does not lead to noticeable performance penalty in practice. On the other hand, this design enables the system to scale, efficiently managing many concurrent snapshots.

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. The scheduler manages the snapshots and coordinates runners to execute on RComps taking locality into consideration. The memory subsystem serializes the committing of the memory ranges only when runners exit. The RComps facilitate the memory subsystem to handle the runners' memory accesses by leveraging the virtual memory hardware to catch and handle the page fault when a runner accesses a page for the first time, following well-known practices [25], [26]. When handling a page fault, the memory subsystem first looks up the page in the runner's snapshot to obtain the version and location information of the page and prevent the runner from accessing memory outside its snapshot. The memory subsystem makes a copy of the page if it is on the local node to protect that version which

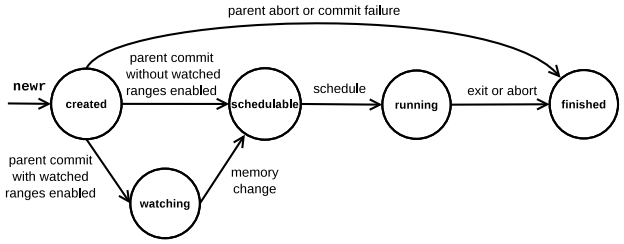


Fig. 4: State transition of the runner and watcher

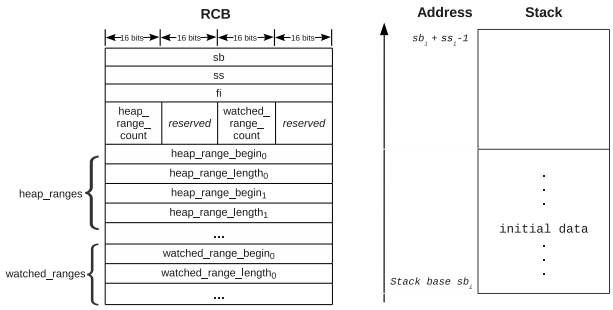


Fig. 5: Runner i's initial stack and RCB

is potentially included in other runners' snapshots, or fetches the page from the remote node through the network.

3.3 Parallelization 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. The 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 large-scale parallel computation, a scheduler to manage the runners, 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. Fig. 4 summarizes the state transitions in a runner's life cycle. The parent runner creates a new runner by executing the `newr` instruction. A "created" runner moves to the "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 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. Because runners are started by the scheduler, 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 such control information. Fig. 5 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. 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 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 4 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. In addition, the scheduler implements the DVM's watcher mechanism (to be discussed in Section 3.3.4).

The scheduler of DVM is a distributed service consisting of two parts: the scheduler master (the master) and the scheduler agent (the agent). There is a single master in a DVM, and there are many 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 construct and access the child runners' stacks. After the parent exits and commits successfully, the scheduler adds the child runners to the runner pool and marks them as 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 a bottleneck in the DVM in practice. Several large-scale systems, such as MapReduce [11], GFS [27], and Dryad [12], follow a similar single-master design pattern.

Together with the design of the scheduler, the design of other components of DVM optimize the workflow to ensure scalability. The snapshot-based memory model of DISA enables massive runner-level parallelism by invoking scheduling and coordination only when a runner starts and commits. The ISA-level lightweight task creation through the `newr` instruction enables programs to create tasks at low cost. The two-version design makes the metadata management fast and efficient so that the memory subsystem can scale to handle many snapshots. These designs ensure that DVM scales well as the processor count and the data size increase. In Section 4, we examine DVM's scalability.

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 that the parallelization mechanism of DVM 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. While a runner may write multiple memory ranges in the SR during its execution, the runner may not want to apply some changes, such as the temporary data, to the global memory space. 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 ranges constructed by its parent runner and heap ranges, and can also read data from I/O channels during its execution. The runner's output is written into its heap ranges and I/O channels if it is needed. Upon completion, a runner can specify whether it wants to commit or abort its changes to the heap ranges. The changes to a runner's stack are always discarded upon completion of the runner. We call this memory usage scheme the SIHO (Stack-In-Heap-Out) model.

To support many-runner parallel execution on the ISA level, the runner creation and dispatching mechanism must be 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, the runner start-up procedure is as follows.

1) The scheduler chooses $RComp(R_j)$ for R_j
2) The scheduler creates snapshot A of R_j's stack range and heap ranges according to RCB_j
3) The scheduler sends A and RCB_j to $RComp(R_j)$ and commands $RComp(R_j)$ to start R_j
4) $RComp(R_j)$ sets R_j's local context according to RCB_j
5) $RComp(R_j)$ starts to execute R_j from fi

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 execu-

tion does not violate dependence relations. The existing approaches have their limitations and constraints: the synchronization mechanisms are not natural or efficient to represent dependence [28], [29]; MapReduce employs a restricted programming model and makes it difficult to express sophisticated application logic [18], [19], [20]; DAG-based frameworks, such as Dryad [12], incur non-trivial burden in programming, and automatic DAG generation has only been implemented for certain high-level languages [16].

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 [30], and sophisticated computation logic can be processed effectively with dynamically scheduled parallelism. 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. We refer the readers to [22] for the discussion of technical details of the watcher mechanism.

3.4 I/O, signals and system services

For simplicity and flexibility, DISA adopts the memory mapped I/O for DVM to operate on I/O channels by accessing memory ranges in AR.

For example, we map the 1-byte memory range at `0x100000200` to the standard input (STDIN) and the range at `0x100000208` to the standard output (STDOUT). The runner can input from STDIN by reading the address `0x100000200` and output to STDOUT by writing to the address `0x100000208`. The DVM translates the memory accesses to these addresses to requests to the RComp for I/O operations. For example, one runner can read one byte from STDIN and write the value to STDOUT by the following instructions.

```
mov:z 0x100000200:b, 0x300001000:b # read one byte from STDIN
mov:z 0x300001000:b, 0x100000208:b # write the byte to STDOUT
```

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 a DVM, many system events may occur. We design a signal mechanism to enable runners to catch the information of events of interest in the system. Signals are notifications of these events. The signal mechanism is built on 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.

To support system functions, such as the I/O and signal mechanisms, several distributed system services cooperate with the scheduler and RComps in DVM. For example, the I/O services on RComps manage I/O channels, and the "system state" service broadcasts "system idle" 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.

3.5 Programming on DVM

Being Turing-complete, DISA instructions are capable of expressing any computational logic. They can also emulate the semantics of instructions in other ISAs. Additionally, we design DISA as an extensible instruction set so that more instructions can be added to DISA if it is sufficiently beneficial to do so. More importantly, the parallelization and dependency control mechanisms along with the snapshot-based memory model provide a convenient and efficient way to design concurrent programs in the "lock-free" style [31].

We use an example to show how to naturally express task dependence and avoid race conditions using watchers. Using the `newr` instruction, we can easily create 2 *sum_runner* runners introduced in Section 2 to calculate sums in parallel. In this example, we design a new runner to add the 2 sums together. The new runner, implemented as a watcher shown in the following code, "watches" the output of the 2 *sum_runner* runners.

```
 1  sum_watcher:
 2  ;read the heap addresses of the depended runners and
 3  ;  the address to store the result from the stack, and stores
 4  ;  them in 0x300001000, 0x300001008, and 0x300001010 (omitted)
 5  ;check whether the depended data are ready
 6  br:e $0:uq, 8(0x300001000), create_self
 7  br:e $0:uq, 8(0x300001008), create_self
 8  ;sum the two integers as the sum_runner
 9  add (0x300001000):uq, (0x300001008), (0x300001010)
10  mov:z $1:q, 8(0x300001010):q            ; exit code
11  exit:c
12  create_self:
13  ;create sum_watcher with the same control information (omitted)
15  exit:c
```

The *sum_watcher* is activated when either of the *sum_runner*s commits and can check their 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 on if either of the two depended runners has not exited and committed. As shown in this example, programmers only need to create the watchers, and the dependences are automatically resolved by the scheduler and watcher-related mechanisms in the DVM.

Although we show the examples in DISA in this paper, compilers can certainly be involved in the programming in the DVM environment and transform programs written in high-level languages into DISA code. With a sufficiently sophisticated compiler, it is also possible to port traditional software to DVM by recompiling them. Providing a compiler and runtime support for a high-level language on DVM is one piece of our ongoing work.

Although one program instantiated as many runners runs in a DVM at a specific time, many DVMs can run concurrently in one cluster by isolating the computing

(a) CCMR (b) TH-1/GZ

Fig. 6: The CCMR and TH-1/GZ supercomputer

resources to ARCs. In the CCMR research testbed (introduced in Section 4), we set up multiple DVMs on the same cluster and many users can develop and run their programs in a time-sharing manner.

4 Implementation and Evaluation

We have implemented DVM and DISA on x86-64 hardware. 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 hardware resources including CPUs and local file systems.

We emulate DISA instructions on x86-64 machines in our current implementation by using binary translation to generate x86-64 code on the fly during the execution of a DISA program. We have implemented and run DVM on multiple platforms as listed below.

- The CCMR research testbed. As shown in Fig. 6(a), the CCMR testbed is constructed for cloud computing research. It is composed of 50 servers with Intel Quad-Core Xeon processors. The servers are connected by 10Gbps and 1Gbps networks. We use 16 nodes connected by the 10Gbps network for the microbenchmarks and comparison experiments, and up to 50 nodes for experiments with large datasets.
- The Amazon Elastic Compute Cloud (EC2). To examine DVM's portability and scalability on public IaaS platforms, we conduct part of the experiments on Amazon EC2 instances of the m1.large type in the us-east-1a zone.
- The TH-1/GZ supercomputer. To examine how DVM performs in a high-performance computing (HPC) environment, we conduct some experiments on the TH-1/GZ supercomputer (Fig. 6(b)) which follows the same architecture of Tianhe-1A [32] ranked currently number 5 in the TOP500 list [33].

The implementations on EC2 and TH-1/GZ also show the portability of DISA in virtualized, heterogeneous and HPC environments. In addition to these 3 platforms, we also implemented and evaluated DVM on an industrial testbed [22]. Although the four platforms have quite different underlying hardware configurations and different Linux kernels, it proves to be straightforward to port DISA to all platforms by adding logic to match specific kernel data structures. Application programs run unchanged at all on four platforms without recompilation.

4.1 Workloads and methodology

We measure DVM's mechanisms with microbenchmarks, compare DVM's performance with Hadoop and X10's, and inspect its scalability. 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 [19], Mars [34], CGL-MapReduce [18] and MapReduce online [35], and has been used extensively by many organizations [36]. X10 represents a class of language-level solutions, including Chapel [21] and Fortress [13], targeting "non-uniform cluster computing" environments [14].

We use several workloads to evaluate DVM's efficiency and scalability. We first design the *prime-checker* which uses 1000 runners to check the primality of 1,000,000 numbers. As the prime-checker is highly parallelizable, we use this arithmetic application to evaluate the scalability of DVM as we add more compute nodes—whether DVM can achieve near-linear speedup. To check DVM's performance on widely used and more complex algorithms, we implement the *k-means* clustering algorithm [37]. The k-means program iteratively clusters a large number of 4-dimensional data points into 1000 groups. This reflects known problem configurations in related work [38], [39], [40]. We run k-means on DVM by scaling the number of compute nodes and the size of datasets to evaluate DVM's scalability, and compare to Hadoop and X10. For more details of the implementation of the prime-checker and k-means programs, we refer the readers to [22]. Additionally, to examine how DVM scales as the overall memory usage grows, we implement and run *teragen* which produces the TearSort [41] input datasets of various sizes.

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, and we write the x86-64-based microbenchmark programs in comparison in the assembly language. Note that the platforms have different performance characteristics. To make the comparison fair, we indicate the platform, use the same compute nodes and network topology and configuration to run comparative experiments, and repeat the experiments to make sure results are consistent.

4.2 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 and efficiency of creating runners, executing runners and memory fetching. In addition, we measure the performance gains from register-based optimization.

TABLE 2: 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 3: 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
	Initialize RComp	0.4
	Update RComp	0.5
	Invoke the runner	0.2
	Commit snapshot	0.9
	Overall	2.4

Virtualization overhead. We design microbenchmarks to measure the overhead of CPU and memory virtualization in DVM. To assess the overhead of CPU virtualization, we measure the arithmetic and logic operation performance by a microbenchmark program that executes 2^{34} *add* operations. Memory is another important subsystem in the DVM, and we shall verify that the distributed memory subsystem 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 microbenchmarks in x86-64 assembly and DISA, and execute the programs in Xen, VMware Player, native Linux, and a DVM on the same physical host. To achieve the best performance, the x86-64 assembly implementations heavily use registers. In the DVM, the binary translator translates DISA instructions into x86-64 instructions during the execution.

Table 2 shows the results of the microbenchmarks. From the results, 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.

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 parent 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 compute node so that we can obtain the completion time of each operation.

Table 3 shows the overhead of creating and executing a runner and the time used in each step. The results show 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.

TABLE 4: Execution time of the reader runner

Configuration	Time / CCMR (second)	Time / TH-1/GZ (second)
2 RComps (Remote memory fetching)	51.05	36.21
1 RComp (Local memory fetching)	6.85	5.63

Memory fetching. We measure the efficiency of the memory operations across multiple physical hosts by a *writer-reader* microbenchmark. The writer-reader microbenchmark program consists of two runners—one, the *writer*, writes 2^{30} 64-bit integers (8 GB in total) into the SR, and the other one, the *reader*, reads these integers from the SR. We run the writer-reader program on DVMs that consist of one or two RComps residing on one or two nodes respectively to check the performance of memory fetching from the local node and the remote node through the network.

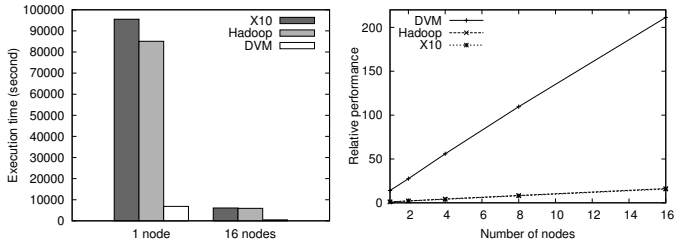
Table 4 shows the results of the writer-reader microbenchmark. In the 1-RComp experiment, it takes 6.85 seconds on CCMR and 5.63 seconds on TH-1/GZ for the reader to read the data. Although providing the advanced snapshot semantics and keeping two versions of the pages as discussed in Section 3.2, DVM exhibits the page fault handling throughputs of 3.1×10^5 and 3.7×10^5 page faults handled per second on CCMR and TH-1/GZ respectively. The additional execution time in 2-RComp experiments is spent on the data transmission through the network. Applying copy-on-write may potentially further improve the performance, which is one direction of our future work.

Register-based optimization. To measure how much performance improvement a program can achieve by using RAMRs, we develop a *loop* microbenchmark since loops are very common in programs. The loop microbenchmark sums 4×10^{10} 64-bit integers in 4×10^{10} iterations. We implement the loop microbenchmark in DISA with and without putting the iterator and accumulator in RAMRs. We also implement and optimize it in assembly using physical registers on Linux for comparison.

Table 5 shows the execution time of loop on both DVM and native Linux on the same compute node. From the results, we can see that, after using RAMRs, loop uses only 18.5% of the time of the other implementation not using RAMRs on both CCMR and TH-1/GZ. The results

TABLE 5: Execution time of loop

Platform	Time / CCMR (second)	Time / TH-1/GZ (second)
DVM, using RAMRs	31.83	27.30
DVM, without using RAMRs	171.81	147.34
Native Linux	31.79	27.28



(a) Execution time on 1 and 16 nodes (b) Speedups as the number of compute nodes grows

Fig. 7: Execution time and speedups of k-means

also show that loop on DVM using RAMRs produces very close performance as the manually optimized one using physical registers on Linux on both CCMR and TH-1/GZ. This indicates that the binary translator can generate high-quality native code and also verifies the low virtualization overhead of DVM.

4.3 Performance comparison

We run the performance evaluation programs with the same dataset on different numbers of compute nodes on the CCMR testbed to compare DVM's performance and scalability with X10 and Hadoop's.

Table 6 shows the time of prime-checker on Hadoop and DVM. We scale the number of compute 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 its advanced instruction set features and memory model.

Fig. 7(a) presents the execution time of k-means on Hadoop, X10 and DVM with 1 and 16 compute nodes. The execution time on Hadoop and X10 is 11 times more than that on DVM on 1 compute node. As shown in this evaluation, DVM quickly exhibits superiority in efficiency as the complexity of application grows. The results on 16 compute nodes show that DVM is at least 13 times faster than Hadoop.

Fig. 7(a) presents that DVM exhibits tremendous speed gains over Hadoop and X10, although Table 6 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 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 Fig. 7(a) reflect DISA's generality, efficiency and scalability.

TABLE 6: Execution time of prime-checker on Hadoop and DVM

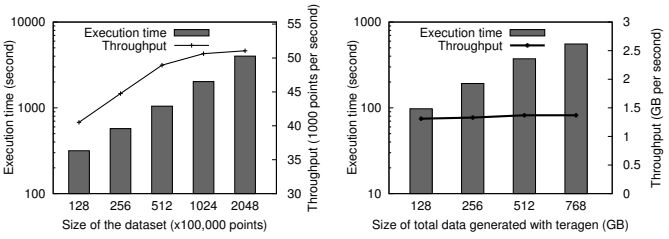
Number of nodes	1	2	4	8	16
Hadoop (second)	16661	8352	4169	2090	1112
DVM (second)	15795	7885	3950	1975	986

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) [14]. As shown in Fig. 7(a), however, the DVM is one order of magnitude faster than both Hadoop and X10. In fact, disk I/O does not necessarily outweigh 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.

Fig. 7(b) shows the speedups for three technologies as we scale the number of compute nodes. All speedups are calculated with respect to the X10 performance on one compute node for each workload. The speedup of the k-means computation on DVM is near-linear to 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.

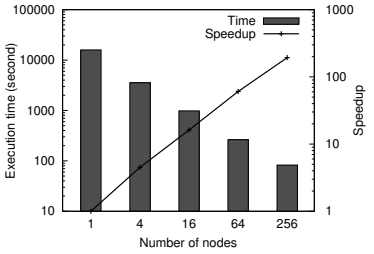
4.4 Scalability

A scalable system may be required to scale with the data size, the processor count, the user population, or a combination of them. As user-scalability is not a goal of DVM, our evaluation focuses on the scalability with the data size and the number of compute nodes. The former also measures the scalability of the memory subsystem.



(a) k-means (b) teragen

Fig. 8: Scalability of DVM with the data



(a) prime-checker on EC2

Platform and # of nodes	Execution time (second)	Speedup
TH-1/GZ		
160	480	12.99
80	822	7.58
40	1578	3.95
20	3128	1.99
10	6233	1.00
CCMR		
50	4011	1.55

(b) k-means on TH-1/GZ

Fig. 9: Scalability with the number of compute nodes

We run k-means on DVM with 50 compute nodes on the CCMR testbed to evaluate DVM's scalability with the dataset size by increasing the dataset from 12,800,000 points to 204,800,000 points. Fig. 8(a) presents the execution time and throughput which is calculated by dividing the number of points by the execution time. The result shows that, on the same number of compute nodes, the execution time grows more slowly than linear growth with the dataset size while the throughput increases. Hence, DVM scales well with the dataset size.

To examine how DVM performs when the total memory usage grows, we run *teragen* on 32 compute nodes to generate data in the SR with increasing sizes. We stress the memory subsystem by generating data of up to 768GB which reaches the capacity of the 32 compute nodes' overall available physical memory. Fig. 8(b) shows the execution time and throughput of teragen. The throughput is approximately the same as the total data generated increases from 128GB to 768GB, indicating DVM's memory subsystem scales well with increasing memory usage. As one direction of future work, we will explore how to unify the physical memory and disks and design a "virtual memory" for DVM.

To evaluate DVM's scalability and also examine DVM's performance in a public cloud, we run the prime-checker on a DVM running on Amazon EC2 with up to 256 virtual machine instances. Fig. 9(a) 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 compute nodes—the speedup is near-linear as we increase the number of instances. Our ability to perform experiments 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 experiments. 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.

4.5 Performance on the TH-1/GZ supercomputer

To examine the performance and scalability of DVM in high-performance computing environments, we run k-means on the TH-1/GZ supercomputer with 10 to 160 compute nodes. The input dataset consists of 204,800,000 points.

Fig. 9(b) shows the execution time and speedup of k-means on TH-1/GZ. The speedup is calculated with respect to the execution time on 10 compute nodes on TH-1/GZ. For comparison, we also show the execution time and speedup of k-means on CCMR with 50 compute nodes in Fig. 9(b). The performance of the DVM on 50 compute nodes on CCMR is between the performance of DVM on 10 and 20 nodes on TH-1/GZ. Although these two platforms have different performance characteristics, the results verify that DVM can deliver efficient performance in different environments. From the results, we can see that when we scale the compute nodes from 10 to 160, the speedup of DVM also increases to 12.99, which indicates that DVM scales well in the supercomputing environment.

The experiments of DVM on TH-1/GZ show that, although originally designed for datacenter environments, DVM is portable to run on HPC platforms, conducting computation efficiently and scaling well. These results also suggest a potential of using cloud system software for high-performance computing.

5 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. created the MapReduce programming model to process and generate large data sets [11]. MapReduce relies on a restricted programming model so that it can automatically run the programs in parallel and provide fault-tolerance while ensuring correctness. Dryad takes a more general approach by allowing an application to specify an arbitrary "communication DAG (directed acyclic graph)" [12]. While these frameworks are successful in large data processing, they have their limitations. The restricted programming model such as MapReduce is not general enough to cover many important application

domains and makes it difficult to design sophisticated and latency-sensitive applications [17], [18], [19], [20]. Building the DAG for Dryad applications is a non-trivial task to the programmers and automatic DAG generation is only seen implemented for certain high-level languages. 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.

As another approach, new languages may help programmers write clearer, more compact and more expressive programs on a cluster. High-level languages, such as Sawzall and DryadLINQ, which are implemented on top of programming frameworks (MapReduce and Dryad), make the data-processing programs easier to design [15], [16]. However, these languages also suffer from the limitation of the underlying application frameworks. Charles et al. design X10 to write parallel programs in non-uniform cluster computing system, taking both performance and productivity as its goals [14]. 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 parallelization and concurrency control mechanism along with the shared memory model.

Different from the frameworks and languages that make the data communication transparent to programmers, message passing-based technologies [42] require that the programmer handle the communication explicitly [43]. 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 transmission among computers transparent to programmers [44]. The DSM systems, such as Ivy [26] and Treadmark [25], combine the advantages of "shared-memory systems" and "distributed-memory systems" to provide a simple programming model by hiding the communication mechanisms [44], 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 multiprocessor (SMM) system and provides a single system image (SSI) on top of workstations connected through an Ethernet network that provides "causally-ordered delivery" [10]. 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 [4], [6]. Both vNUMA and the emerging "inverse virtualization" [45] aim to merge a number of compute nodes to be one larger machine. Different from vNUMA, which is designed to be used on small clusters [10], DVM virtualizes a large number of nodes in large clusters at the ISA level and allows programs to scale up to many logic flows.

6 Conclusion

Cloud computing is an emerging and important computing paradigm backed by datacenters. However, developing applications running in datacenters is challenging. We design and implement DVM—a big virtual machine that is general, efficient, scalable, portable and easy-to-program—as an approach to developing a next-generation computing technology for cloud computing. Giving programmers an illusion of a "big machine", we design the DISA as the programming interface and abstraction of DVM. We evaluate DVM on research 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.

7 Acknowledgments

This work was supported in part by the HKUST research grants REC09/10.EG06, DAG11EG04G and SJTU grant 2011GZKF030902. We thank the Amazon AWS research grant and the Guangzhou Supercomputing Center for the support in the EC2 and TH-1/GZ based evaluation.

References

[1] "Amazon Elastic Compute Cloud – EC2," *http://aws.amazon.com/ec2/*, [8 Oct. 2012].
[2] "Rackspace," *http://www.rackspace.com/*, [8 Oct. 2012].
[3] "Windows Azure," *http://www.windowsazure.com/*, [8 Oct. 2012].
[4] 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 *Proc. of SOSP'03*, 2003, pp. 164–177.
[5] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux virtual machine monitor," in *Proc. of the Linux Symposium*, vol. 1, 2007, pp. 225–230.
[6] C. A. Waldspurger, "Memory resource management in VMware ESX server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, 2002.
[7] R. Buyya, T. Cortes, and H. Jin, "Single system image," *Intl. Journal of High Performance Computing Applications*, vol. 15, no. 2, p. 124, 2001.
[8] 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*, 2009, pp. 124–131.
[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*, vol. 4, no. 1, pp. 1–108, 2009.
[10] M. Chapman and G. Heiser, "vNUMA: A virtual shared-memory multiprocessor," in *Proc. of USENIX ATC'09*, 2009.
[11] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters." in *Proc. of OSDI'04*, 2004, pp. 137–150.
[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. of EuroSys'07*, 2007, pp. 59–72.
[13] 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, [8 Oct. 2012].

[14] 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*, vol. 40, no. 10, 2005, pp. 519–538.
[15] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with Sawzall," *Sci. Program.*, vol. 13, no. 4, pp. 277–298, 2005.
[16] 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 *Proc. of OSDI'08*, 2008, pp. 1–14.
[17] 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*, 2010, pp. 68–73.
[18] J. Ekanayake, S. Pallickara, and G. Fox, "MapReduce for data intensive scientific analysis," in *Fourth IEEE Intl. Conf. on eScience*, 2008, pp. 277–284.
[19] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proc. of the 13th Intl Symposium on High Performance Computer Architecture*, 2007, pp. 13–24.
[20] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-Reduce-Merge: simplified relational data processing on large clusters," in *Proc. of SIGMOD'07*, 2007, pp. 1029–1040.
[21] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *Intl. Journal of High Performance Computing Applications*, vol. 21, no. 3, p. 291, 2007.
[22] Z. Ma, Z. Sheng, L. Gu, L. Wen, and G. Zhang, "DVM: Towards a datacenter-scale virtual machine," in *Proc. of the Eighth Annual Intl. Conf. on Virtual Execution Environments*, 2012, pp. 39–50.
[23] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, pp. 66–76, 1996.
[24] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," in *Proc. of SIGMOD'95*, 1995, pp. 1–10.
[25] P. Keleher, A. Cox, S. Dwarkadas, and W. Treadmarks, "Distributed shared memory on standard workstations and operating systems," in *Proc. 1994 Winter USENIX Conf.*, 1994, pp. 115–131.
[26] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, 1989.
[27] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. of SOSP'03*, 2003, pp. 29–43.
[28] C. Tseng, "Compiler optimizations for eliminating barrier synchronization," in *ACM SIGPLAN Notices*, vol. 30, no. 8, 1995, pp. 144–155.
[29] D.-K. Chen, H.-M. Su, and P.-C. Yew, "The impact of synchronization and granularity on parallel systems," in *Proc. of the 17th Annual Intl. Symposium on Computer Architecture*, 1990, pp. 239–248.
[30] K. Birman, G. Chockler, and R. van Renesse, "Toward a cloud computing research agenda," *SIGACT News*, vol. 40, no. 2, pp. 68–80, 2009.
[31] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proceedings of the 20th annual international symposium on computer architecture*, 1993, pp. 289–300.
[32] X. Yang, X. Liao, K. Lu, Q. Hu, J. Song, and J. Su, "The TianHe-1A supercomputer: its hardware and software," *Journal of Computer Science and Technology*, vol. 26, no. 3, pp. 344–351, 2011.
[33] TOP500 Supercomputer Sites, "TOP500 List," *http://www.top500.org/list/2012/06/100*, [8 Oct. 2012].
[34] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," in *Proc. of the 17th Intl. conference on parallel architectures and compilation techniques*, 2008, pp. 260–269.
[35] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," in *Proc. of the 7th USENIX conf. on networked systems design and implementation*, 2010, pp. 21–21.
[36] Apache Hadoop, "PoweredBy," *http://wiki.apache.org/hadoop/PoweredBy*, [8 Oct. 2012].
[37] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, 1967, pp. 281–297.
[38] H. Jégou, M. Douze, and C. Schmid, "Improving bag-of-features for large scale image search," *Intl. Journal of Computer Vision*, vol. 87, no. 3, pp. 316–336, 2010.
[39] D. Lee, S. Baek, and K. Sung, "Modified k-means algorithm for vector quantizer design," *IEEE Signal Processing Letters*, vol. 4, no. 1, pp. 2–4, 1997.
[40] R. Zhang and A. Rudnicky, "A large scale clustering scheme for kernel k-means," in *Proc. of the Sixteenth Intl. Conf. on Pattern Recognition*, 2002, pp. 289–292.
[41] "Sort Benchmark," *http://sortbenchmark.org/*, [8 Oct. 2012].
[42] Message Passing Interface Forum, "MPI: A message-passing interface standard," *http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf*, 2009, [8 Oct. 2012].
[43] 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.*, 1995, p. 37.
[44] B. Nitzberg and V. Lo, "Distributed shared memory: A survey of issues and algorithms," *Computer*, vol. 24, no. 8, pp. 52–60, 1991.
[45] B. Hedlund, "Inverse virtualization for Internet scale applications," *http://bradhedlund.com/2011/03/16/inverse-virtualization-for-internet-scale-applications/*, [8 Oct. 2012].

Zhiqiang Ma is currently a postgraduate student working towards the Ph.D. degree in computer science and engineering in the Department of Computer Science and Engineering at the Hong Kong University of Science and Technology (HKUST). He received the B.S. degree from Fudan University in 2009. His current research focuses on large-scale distributed computing, data processing and storage systems for cloud computing.

Zhonghua Sheng received the M.Phil. degree from the Hong Kong University of Science and Technology (HKUST) in 2012. He received the the B.S. degree from Harbin Engineering University and the M.S. degree from Harbin Institute of Technology. He conducted research on large-scale parallel computing and storage systems for cloud computing during his study at HKUST.

Lin Gu is an Assistant Professor in the Department of Computer Science and Engineering at the Hong Kong University of Science and Technology (HKUST). He received B.S. from Fudan University, M.S. from Peking University, and Ph.D. in Computer Science from the University of Virginia. His research interest includes cloud computing, operating systems, computer networks and wireless sensor networks.