

DPS: A Distributed Proportional-Share Scheduler in Computing Clusters

Chang-Hao Tsai and Kang G. Shin

Department of Electrical Engineering and Computer Science

The University of Michigan, Ann Arbor

Email: {chtsai,kgshin}@eecs.umich.edu

Abstract—To improve cluster utilization, we propose a *distributed proportional-share* (DPS) CPU scheduling mechanism, which decouples a parallel program’s structural number of nodes (*program specification*) from the maximum resource usage at run-time (*resource specification*). In order to provide performance isolation between programs, we virtualize each node and run programs within virtual machines (VMs), which we call a *VM hosting cluster*. A proportional-share CPU scheduler is then utilized to schedule the execution of programs. On top of this platform, CPU shares are dynamically transferred between hypervisors using either a *fully-distributed* or *bank-assisted* strategy, both of which have been implemented and evaluated. Compared to uniform static resource allocation, DPS is shown to reduce the program completion time significantly.

I. INTRODUCTION

Recently, cluster computing has become a popular means of realizing a growing number of applications. A computing cluster may consist of thousand of nodes connected via high-speed, low-latency networks and provides unprecedented computing power. Parallel programs are frequently used to solve large-scale complex problems. When users submit parallel programs for execution, they usually specify resource requirements, such as the number of computing nodes, amount of memory and disk space. Unfortunately, as the problem size scales up and more detailed models are used, it becomes very difficult to create parallel programs that place balanced loads on all of the allocated nodes.

Computing resources allocated to a parallel program are usually evenly distributed to all nodes, and idle CPU cycles will result if the resource demand is unbalanced. Although the low utilization itself may not be a serious problem in research-oriented clusters, it is important to commercial utility clusters. If users are charged even for unused resources, the cluster would be neither cost-competitive nor attractive to the users.

Instead of optimizing parallel programs over all possible inputs, we would like to enable programmers to decouple the number of nodes (*program specification*) from the maximum run-time resource usage (*resource specification*) required by the underlying applications. In addition to specifying the number of nodes a parallel program requires, the user sets the limit of maximum resource the program can use at run-time. For example, one program can use 4 single-CPU nodes but no more than 2.5 CPUs at any time. Time-sharing CPU schedulers can then be utilized to optimize resource usage.

In order to provide secure and flexible resource allocation and ensure a high degree of performance isolation between

resource-competing programs, we constructed a cluster computing environment called a *VM hosting cluster*, where parallel programs are encapsulated within VMs instead of simply creating processes on each node. The hypervisor is then responsible for scheduling CPU and other resources. VMs are also much easier to migrate between hosts [1].

On this platform, we propose a *distributed proportional-share* (DPS) CPU scheduling mechanism. A proportional-share CPU scheduler is utilized in each hypervisor and each VM is initially given a certain share of CPU time. To deal with the intrinsic workload imbalance, part of the CPU share can be “transferred” at run-time from one process to another. We determine workload imbalance by monitoring CPU usage and the communication between processes. When a process A does not consume all its allocated CPU share and its continuing execution depends on other processes, we transfer part of A’s CPU share to its peer processes. By periodically transferring CPU shares, the CPU share of each process becomes proportional to its relative load (to other processes’). It can also capture dynamic workload shifting.

Compared to uniform static CPU-share allocation, DPS is shown to reduce program response time significantly, while improving cluster utilization. In a feedback control real-time scheduling framework such as FCS [2], DPS reduces the actual execution time of a task without committing more resources, raising priorities, or sacrificing the QoS. Although the actual improvement depends on the real workload, we demonstrate that computing clusters can be utilized much better by separating program specification (required number of nodes) from maximal resource allocation and utilizing the share-transferring scheduler.

Our main contributions are (i) separating the maximum resource usage from program specification and (ii) using the proposed DPS scheduling mechanism to improve both program response time and cluster utilization. Two DPS scheduling strategies, namely *fully-distributed* and *bank-assisted share-exchange*, are proposed and implemented. These schemes not only lower the cost of running a parallel program on the cluster, but also ease the creation of parallel programs.

The paper is organized as follows. We start with the design of DPS in Section II which is followed by the share-exchange models in Section III. The implementation is detailed in Section IV and evaluated in Section V. Section VI reviews some related work and Section VII concludes the paper.

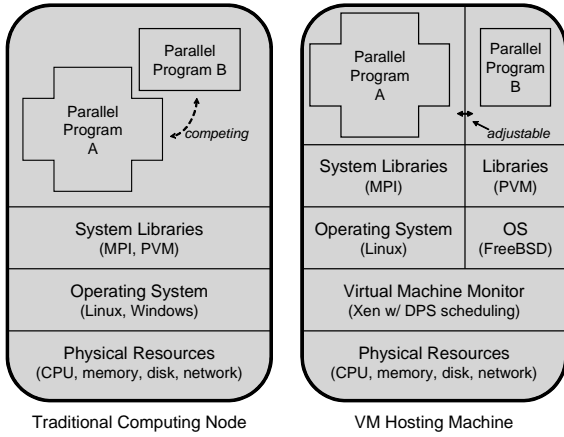


Fig. 1. Comparing traditional computing node and VM hosting machine.

II. DESIGN

We first introduce the VM hosting cluster, where each computing node is transformed into a *host machine* (see Figure 1). We then build the DPS scheduler to provide coordinated resource scheduling across all nodes. This two-tier architecture is similar to a normal operating system (OS), where the lower-tier manages the underlying resources and the upper-tier schedules processes to coordinate usage of resources.

A. VM Hosting Cluster

In a traditional time-sharing computing cluster, while a proportional-share CPU scheduler can guarantee each program to receive a given share of CPU time, a program still competes with others for shared resources such as buffer cache. Moreover, configuration conflicts may even render time-sharing impossible. In a commercial cluster, customers also require the strictest protection to prevent any information leaks.

Instead, we use VMs as a vehicle to execute parallel programs. Each VM can be configured to have a portion of CPU time, memory, disk space, and network bandwidth. Parallel programs execute on a cluster of VMs instead of physical nodes. Standardized VM images can be packaged to provide common program execution environments; customized images can also be made to include proprietary software packages.

In addition to the ability of assigning CPU shares to VMs without tweaking the guest OS scheduler, VMs can also be migrated between host machines. Two or more VMs can be hosted on a single host; they may even be assigned to the same parallel program. With these schemes, idle resources can be re-assigned, and hence, more parallel programs can be hosted/accommodated at the same time.

By running parallel programs on VMs, we can also provide better performance isolation and security measures. From a parallel program’s point of view, it is the only program running within the VM, and hence, no other programs would compete for any of its acquired resources. The guest OS can also be strictly configured to enhance security.

Once resources are virtualized, the specification of a parallel program also needs to be adapted accordingly. Since VMs are dedicated to a single parallel program, the specified number

of nodes a parallel program requires only reflects the structure of the program. When submitting a parallel program, one can separately specify its maximum instantaneous CPU usage. Moreover, he need not break down the CPU allocation into each VM as the DPS scheduler—which we introduce below—can dynamically move resources between all VMs as needed to match the relative resource needs.

This scheme also frees programmers from the job of optimizing the program structure for load-balancing among the nodes. This may sometimes not even be possible because the workload may change dynamically. The load-balancing problem is actually translated into the problem of dynamically allocating resources.

In addition, dynamic resource allocation also enables users with a limited resource budget to execute unmodified large-scale parallel programs. He only needs to submit the program with a low CPU-usage cap and the DPS scheduler will optimize resource allocation. While the program execution is prolonged, the resource utilization is improved.

Although the insertion of a hypervisor may incur some overhead while executing parallel programs, the advantages it provides—namely, more flexible and secure resource allocation and decoupling program structure from resource usage—can better utilize cluster resources and enhance throughput.

B. DPS Scheduler

The DPS scheduler on each host machine utilizes a proportional-share CPU scheduling algorithm, such as lottery scheduling [3] or borrowed virtual time (BVT) scheduling [4], to assign time slots to VMs. We choose a work-conserving algorithm so that idle slots are also distributed proportionally to runnable VMs. The scheduler is also responsible for keeping track of each VM’s CPU shares and actual CPU utilization.

If a VM does not fully utilize its CPU shares, the scheduler may transfer its excess shares to other VMs of the same parallel program. We will henceforth call the other VMs *peer VM*. Each parallel program can employ different strategies to decide the amount of each transfer and its timing. We have designed distributed and centralized share-exchange strategies.

Distributed Strategy: In the *fully-distributed share-exchange* strategy, a scheduler transfers CPU shares directly to another scheduler. The target of the CPU-share transfer is determined by analyzing the network traffic between VMs. Specifically, if a packet from a remote peer VM (VM_r) unblocks the execution a locally-hosted VM (VM_l), the execution of VM_l is said to depend on VM_r , called an *upstream VM* of VM_l . For example, a process in a VM may wait for a result from a process in another VM to continue its execution.

Two VMs can be in the upstream to each other. If both VMs have excess CPU shares, a number of shares may bounce back and forth between them without causing any harm. If only one of them has excess shares, the one with excess shares will initiate a CPU-share transfer to the other. If none of them has excess shares, no share-transfer will occur.

If excess CPU-shares are available for a VM, the DPS scheduler solely determines the amount of share to be trans-

ferred and initiates the transfer. Another DPS scheduler (receiver) can either accept or reject the transfer. The receiver usually accepts any transfer, unless its CPU has been fully booked. When this happens, the receiver recognizes that the host machine is a bottleneck and may seek for migration.

Centralized Strategy: Another approach is to create a centralized CPU-share bank for a parallel program, called a *bank-assisted share-exchange*. DPS schedulers deposit excess resources into the bank of a parallel program. If a VM fully utilized its CPU share and the host machine is not fully booked, the scheduler requests more CPU shares from the bank. A banker process, which is supplied with a parallel program, is responsible for redistributing banked shares to all VMs of the parallel program.

Comparison: The fully-distributed strategy is more robust than the centralized counterpart. A scheduler that fails to participate in share-exchanges only renders a limited amount of CPU shares unavailable. On the other hand, the fully-distributed strategy only transfers shares between peer VMs. The fully-distributed strategy may take more time to propagate excess shares to where they are needed while the bank-assisted strategy can always relay shares by one hop. Also, if the imposed workload fluctuates among distant peer VMs (in terms of their dependency), the bank-assisted strategy is more likely to achieve the global optimum of share distribution, where the fully-distributed strategy may only find local optima.

III. SHARE-EXCHANGE MODELS

We now formalize the cluster and program models and both share-exchange algorithms.¹

A. Cluster Model

We consider a VM hosting cluster of M host machines that are connected to each other via a high-speed network. We assume that there are sufficient network bandwidth, and the execution time of a program is not affected by its placement.

The computing capacity $C_j (j = 1 \dots M)$ of each node is defined as the number of CPUs installed. The virtualization overhead can be incorporated by slightly reducing C_j . For example, to compensate a 5% CPU overhead in virtualization on a dual-CPU SMP node, one can advertise a computing capacity of 1.9 CPU instead of 2. The computing capacity is divided into CPU shares. If a VM can utilize multiple CPUs, the size of a CPU share can range between 0 and C_j . Otherwise, the share size lies between 0 and $\min(1, C_j)$. For heterogeneous clusters, a CPU speed index needs to be established *a priori* to account for relative computing power.

B. Program Model

A program model includes *program specification* and *resource specification*. Among other things, program specification includes the number of nodes N , which can be

¹For simplicity, the subscript for each program and the time index are omitted in the discussion. Subscripts i and j indicate a specific VM and a specific host machine, respectively.

smaller than, equal to, or larger than M . Virtual machines, $VM_1 \dots VM_N$, are created when a program starts running.

Resource specification includes the maximum instantaneous CPU usage W and a share-exchange strategy. Initially, a CPU share $w_i = W/N$ is assigned for VM_i . Other resource specifications such as maximum execution time and utility specifications such as deadline and reward are not used in the model for share-exchange.

Each VM_i may be hosted in different host machines. Distributed node-level schedulers report the incremental CPU time t_i used by VM_i every T seconds and conduct a share-exchange. The reporting process need not be synchronized among all VMs. The CPU utilization u_i by VM_i during the last sampling period is then defined as t_i/T . The DPS scheduler is configured to guarantee u_i is at least w_i if T is large enough and VM_i can fully utilize its CPU time. Note that the scheduler is work-conserving so u_i may be much larger than w_i . An excess share e_i for VM_i is defined as $e_i = \max(w_i - u_i, 0)$. For example, if two CPU-intensive VMs have 0.5 and 0.25 CPU share, the actual CPU utilization will be 0.67 and 0.33, respectively, and there is no excess share. On the other hand, if a VM has 0.5 CPU share and results in 0.1 CPU utilization in the last sampling period, there is 0.4 excess CPU share.

C. Fully-Distributed Share-Exchange

In the fully-distributed share-exchange, a list is maintained to contain the current upstream VMs for each VM_i , and old entries expire after a certain period of time. We assume the list contains P_i entries.

For each VM_i , the DPS scheduler on its host machine redistributes its excess shares to all P_i upstream VMs at the end of each sampling period. VM_i can also withhold part of the excess shares to leave some room for workload fluctuation since a VM does not proactively ask for shares from its peer VMs (it is also more difficult to identify downstream VMs with excess shares.) Given a withholding factor $wh_i (0 \leq wh_i \leq 1)$, the number of shares s_i that the scheduler will send to each upstream VM is calculated as:

$$s_i = \frac{e_i \cdot (1 - wh_i)}{P_i}. \quad (1)$$

Note that when the withholding factor wh_i is set to $1/(P_i + 1)$, VM_i withholds the same amount of share as it will transfer to each upstream VM. In the above example, if the VM having 0.4 excess CPU share has 3 upstream VMs and withholds $1/(P + 1)$ of excess shares, it will initiate a share transfer of 0.1 share to the 3 VMs and, if all transfers are successful, retain 0.2 CPU share at the end of re-distribution.

D. Bank-Assisted Share-Exchange

In the bank-assisted share-exchange, all excess shares are transferred to the bank. Otherwise, if the capacity of the host machine is not fully allocated, the DPS scheduler notifies the bank of the CPU share required to match a VM's allocation to its utilization. This amount is also limited by the remaining share of the host machine r_j . We call this amount *the share shortage* f_i , which can be written as $f_i = \min(u_i - w_i, r_j)$.

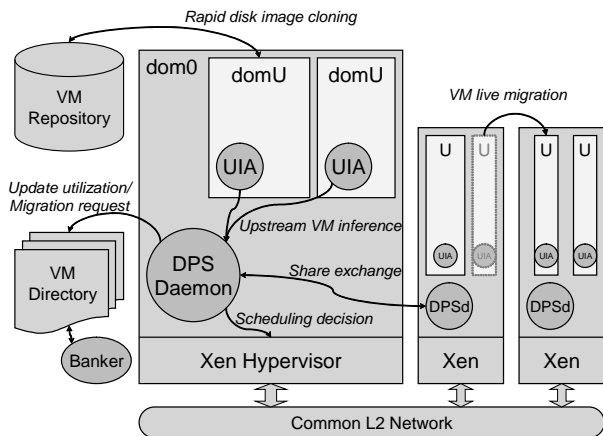


Fig. 2. The architecture of a VM hosting cluster.

For example, a VM having 0.5 CPU share but utilizing 0.67 CPU has 0.17 CPU-share shortage.

The banker process redistributes all the excess shares every B seconds. Assuming the bank has a balance of E shares and $F = \sum f_i$ share shortage, the shares s_i that the banker process will send to each VM are calculated as:

$$s_i = \max(f_i, f_i \cdot \frac{E}{F}). \quad (2)$$

For example, if the banker process has a total of 0.3 excess share and a total share shortage of 0.6, the VM that needs 0.17 share will be given 0.085 share. Excess shares that had not been redistributed are kept in the bank for the next distribution cycle. VMs do not withhold shares in this scheme because they can explicitly ask the banker for more shares.

IV. IMPLEMENTATION

We implemented the VM hosting cluster and DPS scheduler on Xen with its BVT CPU scheduler because of its low overhead and live-migration capability. We first briefly discuss the VM directory and then detail the DPS daemon and the upstream inference system. An overview of the VM hosting cluster implementation is illustrated in Figure 2.

A. VM Directory

When transferring shares, the DPS schedulers and the banker processes only know the address of the share-receiving VM, but not the address of the host machine that hosts the VM. To deal with this issue, we create a VM directory to provide a mapping between the IP address of a VM, its host machine’s name/IP address, and the Xen domain ID of the VM. The VM directory also serves as the bank to store excess CPU shares of each parallel program. Since banker processes require very little resource, any reasonable hardware can host the VM directory and many banker processes easily.

B. The DPS Daemon

At the core of the DPS scheduling is the DPS daemon running within the host OS (domain 0) on every host machine. Its function includes creating VMs, interacting with the BVT scheduler within Xen, reporting utilization to the VM

directory, receiving the upstream inference result from hosted VMs, and conducts share exchanges. We implemented it in C and use the `xc` library to control the scheduler.

In the fully-distributed mode, the DPS daemon looks up the host machines of each upstream in the VM directory and connects to the remote DPS daemons directly. It may also receive CPU shares from remote DPS daemons at any time. In the bank-assisted mode, it sends and receives shares to and from the banker process. After each successful share-exchange, scheduling parameters are changed to reflect the new resource allocation.

C. Upstream Inference

In order to identify the upstream VMs, we need to correlate the packets received and the effect to the processes running in a VM. Every packet received by a hosted VM is actually redirected via the host OS, which re-maps the received packet and sends a virtual IRQ to the receiving VM via a Xen event channel. The event may wake up the guest OS if it was blocked, and the guest kernel receives the packet from a virtual network interface. Standard protocol processing is then applied and wake up processes, if necessary.

Inferring upstream VMs in the host OS or hypervisor is VM-neutral and more generic. However, we can only learn the virtual processor’s state (running or blocked) but not the parallel process’ state because not every incoming packet to a blocked parallel processes will wake it up. For example, data packets may not wake up the receiving process if it was not actively waiting for the specific socket. On the other hand, a data-less TCP ACK packet may indicate a successful data transmission or a connection establishment, and unblocks a process. Unless the protocol is well understood, it is very difficult, if not impossible, to infer dependency outside a VM.

We instrument a guest Linux kernel to infer upstream VMs. After a packet is delivered to user space, the kernel will wake up all processes waiting for it. The sending host’s ID is passed with wakeup functions. If the receiver process isn’t running nor already in the run-queue, we declare the sending host is an upstream VM, and report its IP address and port number. A user-space program, the upstream inference agent (UIA), is then proxies such information back to the DPS daemon.

V. EXPERIMENTAL EVALUATION

We first introduce the testbed setup and benchmark programs and then evaluate the overhead and effectiveness of the DPS scheduling.

A. Testbed Setup

We constructed a testbed of 4 nodes to evaluate the performance of the proposed VM hosting cluster and the DPS scheduler. The benchmark is a PVM parallel program called `patterns`, which was developed for the Virtuoso project [5] and models after bulk-synchronous parallel (BSP) style applications. The program can be configured to run with any number of nodes and produce various topologies commonly used by parallel programs. The length of program execution

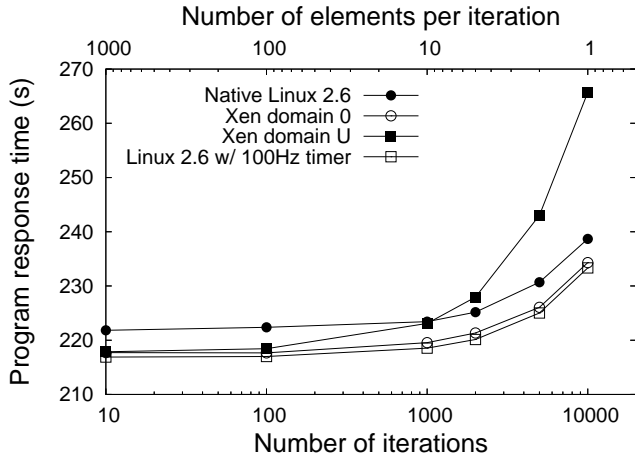


Fig. 3. Xen virtualization overhead.

is controlled by the number of iterations. In an iteration, each node receives a message from one of its neighbors, repeats a simple computation for a number of elements, and sends a message to one of its topological neighbors.

The original program can only generate a uniform amount of computation workload on nodes. To model unbalanced workloads, we modified the source code such that for node i ($i = 1 \dots N$), the amount of computation is scaled to $1/i$ of the specified amount. We also created another serial program `eatcpu` to simulate a single-node CPU-intensive application, which merely consumes all CPU time available to it.

B. VM Hosting Overhead

We first examined the overhead associated with the execution of parallel programs in virtualized environments. The `patterns` program was executed in three different modes: *native*, *Domain-0*, and *Domain-U*. In native mode, the program was executed in Linux without any virtualization. In *Domain-0* and *Domain-U* modes, the program was executed in domain 0 and domain U, respectively. In all cases, no other program was competing for any resource. We configured `patterns` to create a 4-node workload with the all-to-all topology, and each VM was hosted in a different host machine. We fixed total computation demand but changed the number of elements at each iteration to generate different amounts of network I/O (the fewer number of elements per iteration the more network I/O intensive). The program response times in different modes are compared and plotted in Figure 3.

At first, the results didn't match our expectation. Although running in domain U is slightly slower than running in domain 0, the performance in native mode was actually *worse* than running in either case. Only when network I/O is frequent, the program response time was longer in domain U, where the I/O virtualization overhead was dominant. The explanation of this counter-intuitive result is a difference in the kernel timer frequency. We lowered the frequency from 1000Hz (default in Linux 2.6) to 100Hz (default in both Xen domain 0 and U) and the program response time was decreased by 2%. With this fix, we observed that the CPU virtualization overhead is very

small; it is about 0.4% and 0.6% for domain-0 and domain-U mode, respectively. A longer network packet processing path raises the overall overhead to 14% in domain-U mode [6].

Moreover, it took only 18 seconds to boot up a VM, where only 2.5 seconds CPU time was used. We concluded that the virtualization overhead is small, and an optimized kernel configuration can make a performance improvement.

C. DPS Scheduling

The strength of the DPS scheduling can best be shown in a resource-competing environment with unbalanced workloads. The testbed is set up in such a way that 50% of CPU time on each node has already been booked, and only the remaining 50% of CPU time is available for competition. In the absence of competition, even a program with a small share of CPU time can use a full CPU under a work-conserving scheduler. We deployed `eatcpu` in domain 0 of each host machine to emulate this immovable background workload.

On top of this background workload, we executed `patterns` to create a 4-node workload with the linear topology, which is chosen so that we may also compare the fully-distributed and bank-assisted share-exchange strategies. The share-exchange was configured to be performed once every 5 seconds. Conducting the share exchange too often would make the two strategies less distinguishable. On the other hand, infrequent share-exchanges would make the potential benefits of DPS less visible. For the fully-distributed strategy, we set the withholding factor wh_i to $1/(P_i + 1)$.

The `patterns` were submitted with a range of resource specifications, from 0.1 to 0.5 CPU per node, to see how DPS handles different scenarios and program response times are plotted in Figure 4. Without DPS, the CPU shares were fixed and evenly distributed on all host machines. The program response time gets prolonged when fewer CPU shares were given. Note that a 0.1 CPU share only means that at least 10% CPU time is available for this specific VM, and therefore, the response time was not 5 times that of executing with 0.5 CPU share. With an initial CPU share of less than 0.5 per node, there was some unallocated CPU time that DPS can exploit. After a few share-exchanges, CPU shares were redistributed to match the workload.

From Figure 4, we can easily see that DPS significantly reduces the program response time. The user can submit a program with less CPU time budget (*e.g.*, a total of 1.2 CPUs) while having a response time very close to a more costly submission (*e.g.*, a total of 2.0 CPUs). When a total of only 0.4 CPUs is allocated, DPS with the bank-assisted share-exchange strategy cuts down the response time by nearly 38%.

The performance of DPS also depends on the underlying share-exchange strategy. The bank-assisted share-exchange strategy can redistribute the excess shares faster in this topology. This effect is pronounced, particularly when only 0.1 CPU-share was allocated to each node initially. In this case, the most demanding node 1 has enough room to accommodate excess shares from the least utilized (also the farthest) node 4. Therefore, the benefit of the banker process can be seen

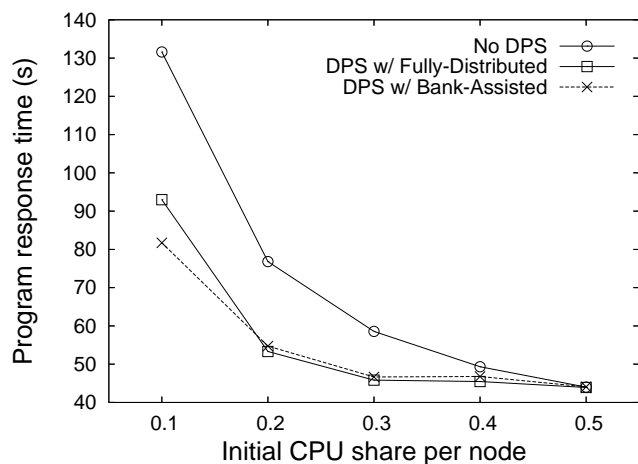


Fig. 4. DPS reduced the program response time by up to 38%.

easily. We also repeat the same configuration with the all-to-all topology. Since all nodes can be reached from each other in one hop, the benefit of the banker process disappeared. We also monitored the CPU usage of the DPS daemon and found the overhead of DPS to be negligible.

VI. RELATED WORK

While VMs have been considered in distributed and grid computing [7], [8], the location of, and resource assigned to VMs are mostly fixed during execution except the VNET in Virtuoso, where VMs spread in a wide area are migrated by matching traffic intensity with available network bandwidth. The peer relationship is inferred by monitoring the traffic volume. On scheduling VMs, VSched utilized a control daemon tweaking real-time scheduler parameters to schedule VMs running as processes in Linux [9].

Gang scheduling, where busy waiting was suggested instead of blocking at synchronization in multiprocessor systems, is also used in clusters [10], [11]. In addition, multiple implicit scheduling approaches are proposed to synchronize the execution of a parallel program across local schedulers using activities such as message send and receive, which imply synchronization [12], [13], [14]. These approaches use a combination of strategies to determine what to do while waiting for messages and when messages have been received.

Proportional-share schedulers, such as lottery and BVT scheduling, are used to provide fairness in a resource-competing environment [3], [4]. While proportional-share scheduling has also been applied to a network of workstations with implicit scheduling [15], [16], its use was limited to support fairness across a cluster.

Separating the program structure from resource management is actually not totally new, either. In *scheduler activations*, users can have a large number of user-level threads to represent the program structure while the kernel allocates processors to user programs [17]. Although there is no load-balancing aspect in scheduler activations, it also frees programmers from the difficult job of gaining maximal benefits from the underlying hardware. Our design provides a similar separation to parallel programs.

VII. CONCLUSIONS

The use of VM enables each hosted parallel program to be executed in a dedicated environment while cluster administrators can easily move resources around. In addition, the proposed DPS scheduler makes it possible to decouple program and resource specifications of a parallel program, and is experimentally shown to yield considerable benefits. Coupling a VM hosting cluster with DPS creates a flexible platform for many scientific and engineering applications and resource utilization is also improved. We believe similar approaches can be applied to other resources as well and a complete cluster OS can be defined and grown out of the concept of VM hosting cluster proposed here.

REFERENCES

- [1] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum, "Optimizing the migration of virtual computers," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 377–390, 2002.
- [2] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao, "Feedback control real-time scheduling: Framework, modeling, and algorithms," *Real-Time Systems*, vol. 23, no. 1, pp. 85–126, July 2002.
- [3] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: Flexible proportional-share resource management," in *OSDI*, 1994.
- [4] K. J. Duda and D. R. Cheriton, "Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler," in *Proc. of the 17th ACM SOSP*, 1999.
- [5] A. Gupta and P. Dinda, "Inferring the topology and traffic load of parallel programs running in a virtual machine environment," in *Proc. of the 10th Workshop on Job Scheduling Policies for Parallel Processing*, 2004.
- [6] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the xen virtual machine environment," in *Proc. of the 1st ACM/USENIX VEE*, 2005.
- [7] R. Figueiredo, P. Dinda, and J. Fortes, "A case for grid computing on virtual machines," in *Proc. of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003)*, May 2003, pp. 550–559.
- [8] B. Calder, A. A. Chien, J. Wang, and D. Yang, "The entropy virtual machine for desktop grids," in *Proc. of the 1st ACM/USENIX VEE*, 2005.
- [9] B. Lin and P. A. Dinda, "VSched: Mixing batch and interactive virtual machines using periodic real-time scheduling," in *Proc. of the 2005 ACM/IEEE Supercomputing*, 2005.
- [10] E. Frachtenberg, F. Petrini, S. Coll, and W. Chun Feng, "Gang scheduling with lightweight user-level communication," in *Proc. of 2001 Int'l Conference on Parallel Processing, Workshop on Scheduling and Resource Management for Cluster Computing*. Valencia, Spain: IEEE, 2001.
- [11] A. Hori, H. Tezuka, and Y. Ishikawa, "Highly efficient gang scheduling implementation," in *Proc. of the 1998 ACM/IEEE Supercomputing*. IEEE, 1998.
- [12] S. Agarwal, A. B. Yoo, G. S. Choi, C. R. Das, and S. Nagar, "Co-ordinated coscheduling in time-sharing clusters through a generic framework," in *Proc. of the 2003 IEEE Int'l Conference on Cluster Computing*, 2003.
- [13] G. S. Choi, J.-H. Kim, D. Ersoz, A. B. Yoo, and C. R. Das, "Coscheduling in clusters: Is it a viable alternative?" in *Proc. of the 2004 ACM/IEEE Supercomputing*. IEEE, 2004.
- [14] P. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien, "Dynamic coscheduling on workstation clusters," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (IPPS/SPDP'98)*. London, UK: Springer-Verlag, 1998, pp. 231–256.
- [15] A. C. Arpaci-Dusseau, "Implicit coscheduling: Coordinated scheduling with implicit information in distributed systems," *ACM Trans. Comput. Syst.*, vol. 19, no. 3, pp. 283–331, 2001.
- [16] A. C. Arpaci-Dusseau and D. E. Culler, "Extending proportional-share scheduling to a network of workstations," in *Int'l Conference on Parallel and Distributed Processing Techniques and Applications*, June 1997.
- [17] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: Effective kernel support for the user-level management of parallelism," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 53–79, 1992.