

Illinois Cluster Manager

Lukasz R. Lempart and James S. Pike

Abstract

Falling prices of commodity hardware and high bandwidth communications infrastructure have made locally distributed computing appealing and affordable. Heterogeneous elements in the cluster, however, introduce complexity to the problem of effectively allocating the full processing capacity of the system. Ideally, users want to issue commands to the pool of resources and they only care how quickly their results will be ready, not where or how their job is processed. Imbalances in node load can cause some processors to be overloaded, while others remain underutilized or idle.

The Illinois Cluster Manager (ICM) endeavors to implement a prototype end-to-end solution that supports an arbitrary set of applications, operating systems, and hardware configurations exposing an extensible yet full featured API that can support a variety of load balancing and decision making modules. By designing ICM with modularity in mind, we hope that others will be able to implement new, experimental load balancing algorithms inside the provided framework. This contribution will enable evaluation of new algorithms without the load balancing researcher having to deal with the intricacies of implementing preemptive migration and remote job invocation.

Section 1: Introduction

Falling prices of commodity hardware and high bandwidth communications infrastructure have made locally distributed computing appealing and affordable. Heterogeneous elements in the cluster, however, introduce complexity to the problem of effectively allocating the full processing capacity of the system. Ideally, users want to issue commands to the pool of resources and they only care how quickly their results will be ready, not where or how their job is processed. Imbalances in node load can cause some processors to be overloaded, while others remain underutilized or idle; many research groups have investigated the rational and equitable use of this available computational power [1] - [4].

In order to facilitate the seamless interoperability of a heterogeneous cluster, processes must be abstracted to have no notion of the underlying hardware on which they execute. Virtualization presents a solution to this difficult problem by allowing processes to be completely encapsulated in a private environment. These sandboxed execution runtimes enable arbitrary code to be executed with no knowledge of the underlying system.

Based on this intuition, the Illinois Cluster Manager (ICM) endeavors to implement a prototype end-to-end solution that supports an arbitrary set of applications, operating systems, and hardware configurations exposing an extensible yet full featured API that can support a variety of load balancing and decision making modules. By designing ICM with modularity in mind, we hope that others will be able to implement new, experimental load balancing algorithms inside the provided framework. This contribution will enable evaluation of new algorithms without the load balancing researcher having to deal with the intricacies of implementing preemptive migration and remote job invocation.

The remainder of this thesis is organized as follows. Section A.2 provides background information and discusses related works. Section A.3 highlights the motivation for this work, and Section A.4 presents the system architecture of the Illinois Cluster Manager. Section A.5 explains the methodology for measuring the performance and overhead of various ICM configurations. Section A.6 presents a detailed discussion of the performance of ICM. Finally, Section A.7 reflects on the lessons learned from this project and suggests directions for future work.

Section 2: Background and Related Work

This section provides a brief overview of material related to ICM. We first discuss concepts related to the virtualization of resources, then follow with a survey of cluster management implementations and conclude with an introduction to load balancing.

Section 2.1: Virtualization

Virtualization refers to the computing technique of abstracting the physical characteristics of hardware into a logical representation. Virtual machine technologies allow running multiple guest virtual machines on a single physical box. Each virtual machine provides a secure, sandboxed environment for processes to run within it. Huang et al. discuss the tradeoffs of using virtual machines for high performance computing applications in a cluster environment [5]. Their work claims that virtualization overhead can be overcome by introducing direct bypasses to hardware devices for operations such as network communication and memory I/O. Furthermore, Huang contends that the ease of management, opportunity for customized operating systems, and enhanced system isolation make up for the virtualization overhead. Huang presents performance results demonstrating that virtualized applications can achieve almost the same performance as those running in a native, nonvirtualized environment.

Process migration refers to the act of transferring a process between two machines [6]. Migration enables dynamic load distribution, fault resilience, and data access locality. A common goal of migration is to leverage additional processing power on underutilized nodes. In addition, migration can allow resource sharing when a particular node provides a unique resource to the system. Since they can be relocated transparently to prevent interruption or data loss, long-running applications benefit from migration. In order to successfully migrate a process, the

system must be able to export and import process state, accurately identify the process along with all of its resources, and clean up the process's nonmigratable state. Nonmigratable state includes local process identifiers, local time, and pending messages yet to be delivered.

QEMU is a machine emulator that runs unmodified operating systems and applications on top of a host operating system transparently to the guest [7]. QEMU implements a dynamic translator to perform runtime conversions of the target CPU instruction set into the host machine instruction set. The resulting conversion is stored in a translation cache for reuse. To perform dynamic translation, QEMU first splits each target CPU instruction into a few simpler instructions called micro-operations. These micro-operations are implemented by a small piece of C code that is then compiled by GCC into an object file. The object files are used to create a dynamic code generator that, in real time, translates hardware instruction sets, allowing code to run on an arbitrary platform without recompiling. Additionally, QEMU supports the creation of virtual FAT disk images from a directory to allow access to a file system without exporting it via a network file system. The KVM extension, which we use in our implementation, provides hardware optimization and live migration over the network [8], allowing for the implementation of transparent preemptive migration.

Section 2.2: Cluster Management

Denali is an attempt to safely execute many independent, untrusted server applications on a single physical machine through the use of paravirtualization techniques [9]. Paravirtualization involves selectively modifying the virtual architecture to enhance scalability, performance, and simplicity. The Denali architecture is based on the x86 instruction set; however, Denali modifies existing instruction semantics, adds virtual registers, modifies interrupt delivery, and eliminates

virtual memory to improve scalability and performance. To improve virtual machine scalability, Whitaker introduces a virtual idle instruction, similar to the x86 halt instruction, which allows a virtual machine to yield control of the processor until more work arrives. Furthermore, Denali modifies the semantics of interrupts to signify the occurrence of an event while its context was switched out. This modification allows interrupts to be queued and delivered asynchronously in batches whenever the target virtual machine is scheduled. In order to reduce machine complexity, Denali does not expose virtual memory. Instead, programs are constrained to use a single address space. Another optimization, implemented by Denali, is the removal of the bootstrapping process and simply loading Virtual machine images into memory. The number of supported devices is also minimized to include only a network interface card, a serial device, a timer, and a generic keyboard. Although Denali is able to achieve high performance and scalability, we believe that the change in basic system semantics and the inability to run arbitrary code in a heterogeneous environment are not tolerable, and show that the execution overhead is amortized by long-running processes. The elimination of bootstrapping, however, is a valuable optimization that does not place undue constraints on the user.

Condor is a distributed scheduling system that operates in a networked workstation environment [3]. The design attempts to maximize the utilization of workstations with minimal interference between the jobs that it schedules and the activities of the workstation owners. Condor attempts to find idle workstations and schedules background jobs to run on these nodes. When the owner of the node resumes activity, Condor checkpoints the remote job and migrates it to another idle workstation. Creating a checkpoint of a program involves saving all the state of the current execution including the text, data, bss, and stack segments of the program, the registers, the status of open files, and any outstanding messages to the Condor system controller.

Condor employs the Up-Down algorithm [3], which trades off the remote execution time users have received with the time they have waited to receive them by maintaining a schedule index for each workstation. The priority by which workstations are awarded remote execution time is determined by the value of its index.

PVM, or parallel virtual machine, proposes using message-passing to allow heterogeneous networks of parallel and serial computers to be programmed as a single computational resource that appears to the programmer as a distributed-memory virtual computer. In order to account for unpredictable variability in the load of individual processors and the network, Casas et al. add three extensions to PVM that allow the system to dynamically migrate components of the active application between the workstations in the network: Migratable PVM (MPVM), user level process PVM (UPVM), and Adaptive Data Movement (ADM) [10]. In MPVM, a daemon process flushes all messages of the process to be migrated. Then, a skeleton process running the same code as the migrating process is started on the target node. The state of the currently running process is transferred via a TCP connection and is loaded into the skeleton. Finally, the skeleton process is restarted and all other processes are informed of this process's new id for message delivery. This design allows for transparent, asynchronous migration that only blocks other processes of the application which communicate with this process. In UPVM, the system is exposed to a set of migratable entities smaller than processes, allowing load redistribution at a finer granularity. The user level process, which UPVM implements, merges the characteristics of threads and processes, and follows the same general algorithm for migration as MPVM. The final approach, ADM, trades migration transparency for detailed information that can improve load balancing and performance. When events demand adaptive load redistribution, all processes of the application must participate.

Since the program is aware of the data semantics of its work, it can better analyze the effects of heterogeneous nodes than either MPVM or UPVM.

The distributed resource management (DRM) system architecture is organized in a five-layer software stack to accommodate matching demand with supply across distributed computing systems [11]. The lowest layer, the resource layer, deals with the node operating systems and performs low level monitoring of system state. The clustering layer implements system primitives necessary to build execution environments within a cluster. The execution environment layer creates and monitors job execution within the cluster. The policies and algorithms that are used to match resource-seeking jobs to available resources are implemented in the demand management layer. The topmost layer, the metacomputing layer, exposes the lower layers to exterior networks.

Section 2.3: Load Balancing

A variety of solutions have been proposed to address the issue of efficiently allocating the processing capacity and memory resources of a locally distributed system in which users submit tasks to arbitrary autonomous nodes connected via a communication network. Ideally, a load balancer transfers tasks from heavily loaded systems to lightly loaded nodes where the job will experience improved performance. A common measure of performance is the response time of a task, which is the time elapsed between the initiation and completion. Our goal is to minimize the average response time of the system.

When evaluating a load distributing algorithm, one must consider the definition of load on the system, centralization, preemption characteristics, transfer policies, selection policies, and system stability. Load indicates the current consumption of resources on a particular node. Load

balancing algorithms commonly employ a load index to predict the performance of a new task on a particular node. Common load indexes include the length of the CPU queue, the average CPU queue length over time, the amount of available memory, the context-switch rate, the system call rate, and CPU utilization. Kunz experimented with the effectiveness of various combinations of workload descriptions and found that the number of tasks in the run queue of the processor held the most influence over the behavior of the system [12]. Furthermore, Kunz found no improvement when multiple load indexes were used to predict the behavior of the system.

Centralization refers to the notion of which node in the system makes the decision of where tasks should run. Algorithms with a high degree of centralization allow for more sophisticated decision policies at the cost of a potential performance bottleneck. Load balancing algorithms can be either preemptive or nonpreemptive when transferring tasks throughout the cluster. A preemptive system allows partially executed tasks to be relocated to another node in the group, while nonpreemptive designs only distribute tasks that are still waiting to begin execution. Stability, in the context of load balancing, refers to the state of an algorithm repeating useless actions indefinitely. One example of unstable behavior is infinitely migrating a task between two lightly loaded nodes without any forward progress with respect to the execution of the task.

A load distributing algorithm typically has four main components: a transfer policy, a selection policy, a location policy, and an information policy [13]. The transfer policy determines whether a node is in a suitable state to participate in a task transfer. Frequently, load balancing algorithms use either a load index threshold or relative load indexes to determine the classification of nodes. The selection policy determines which task should be migrated. General considerations for the selection policy include the overhead of relocation, the remaining

execution time, and the location dependence of each task. The location policy defines how nodes find a suitable transfer partner for task migration. The information policy describes what information will be gathered on each node, how frequently it will be updated, and who will receive it. Common designs consist of demand driven policies where node state is only collected when requested by another operation; periodic policies, where information is coalesced on an elapsed time basis; and state change driven policies, where nodes broadcast new information after their state changes by a predefined degree.

Researchers have proposed and analyzed many techniques for effectively load balancing a distributed system. Harchol-Balter and Downey consider whether preemptive migration or remote execution is necessary for CPU load balancing in networks of workstations running predominately CPU-bound jobs [2]. They find that optimal performing migration policy is heavily tied to the expected remaining lifetime of a job and that preemptive migration outperforms nonpreemptive migration. They argue that nonpreemptive migration is often unable to identify long-running jobs before they begin executing, which inflicts additional delays on other jobs on a given node. Preemption allows the system to correct these oversights later in the job's life and improves performance despite substantial memory and network traffic overheads.

In [1], Arredondo et al. propose a user-supervised processor allocation scheduler dubbed the LBS (load balancing system). In their design, an information subsystem is in charge of collecting and maintaining global information about each node in the cloud. The decision subsystem is responsible for evaluating which is the most suitable node for the execution requested by the user. The execution subsystem executes the user request on the node designated by the decision subsystem. Contrary to [12], [1] suggests that the mean number of processes waiting for the CPU cannot be generically used as a load metric. The paper correlates I/O traffic

in the node, free memory size, and the number of I/O packets being transferred over the network interface with the response time for a job. Furthermore, [1] restricts process migration to be nonpreemptive and requires the cluster to be homogeneous in hardware.

Section 3: Motivation

The motivation for the Illinois Cluster Manager is the observation that many theoretical papers suggest that preemptive task migration is necessary for optimal performance; however, the majority of public cluster management implementations do not support preemption, place limiting restrictions on the composition of cluster hardware, or offer support for preemption only in special environments or with specific language support. Through the use of machine virtualization, ICM offers preemptive, heterogeneous, and transparent support of task migration for purposes of load balancing and system management. This prototype implementation demonstrates the feasibility of such a system and provides initial measurements of overhead and potential benefits. The modularity of our design allows future users of ICM to easily implement new, more sophisticated algorithms for load balancing and remote invocation.

We assume all cluster hosts have access to some form of a shared file system, which contains input files, executables, and a storage location for process output. Furthermore, we design ICM with the assumption that processes running on the cluster will be long-running, CPU-bound jobs without message passing or network communication.

Section 4: System Architecture

The Illinois Cluster Manager is a joint project between James Pike and Lukasz Lempart. This section primarily deals with the implementation of the statistics daemon, election daemon, migration manager, KVM manager, clustering manager, load balancer, decision engine, and shell.

Section 4.1: Architectural Overview

To frame the following discussion, we first highlight the overall architecture of the Illinois Cluster Manager design. Figure 1 demonstrates our component-level system architecture. To maximize portability and extensibility, ICM is composed of three main software layers. At the bottom, a set of KVM/QEMU specific drivers control interaction with the virtual machines through the use of message queues. These drivers expose a predefined interface to the upper layers for the instantiation of jobs and migration of processes. By conforming to the agreed upon interface, another virtual machine or process abstraction can replace KVM with minimal changes to the remainder of the system.

Above the virtual machine drivers lies the clustering manager layer. Within the clustering manager, ICM constructs a logical view of the network as a whole, and controls the creation and maintenance of the cluster. Through the use of heartbeating and a modified version of the bully algorithm leader election protocol [14], the clustering manager monitors the health of each node in the network and nominates a coordinator to make administrative decisions. Attached to each heartbeat message is a set of node statistics gathered by the statistics daemon. Higher layer software can use the gathered statistics to make intelligent decisions about the state of the cluster.

In addition, the clustering layer exposes an API to the shell and load balancer modules that supports starting tasks on specific nodes and requesting migrations between hosts.

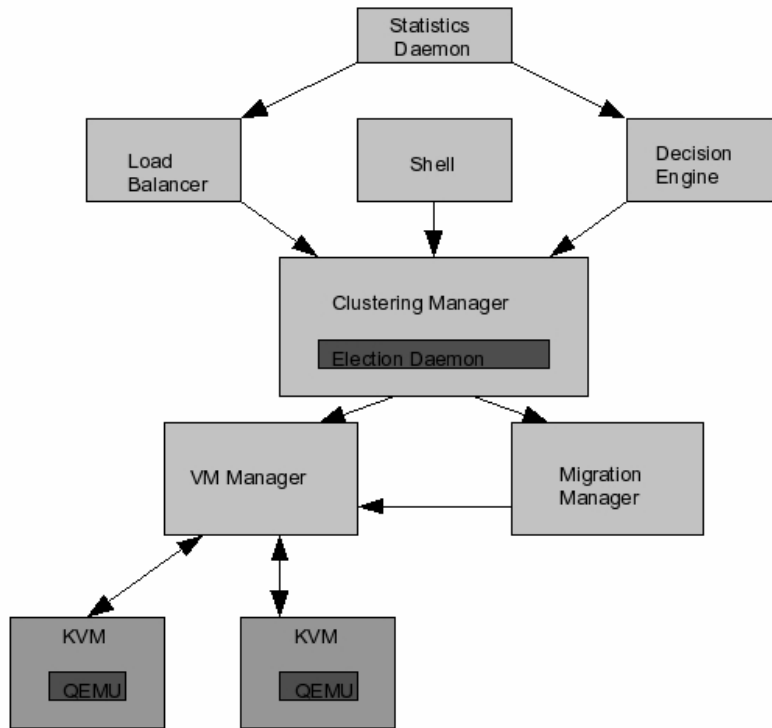


Figure 1: ICM component architecture. All components run on each node with the exception of the load balancer, which only runs on the coordinator node.

The top layer consists of the user shell, decision engine, and load balancer. The shell is responsible for accepting user input and making the appropriate translation into the clustering layer API. Based on the currently installed decision algorithm, the decision engine uses the node state information provided by the clustering layer to determine the optimal node on which to start the next job. Similarly, the load balancer analyzes the current system state and determines when to relocate a job. In keeping with our stated goal of modularity, the choices of both the load balancer and decision engine algorithms can be modified through a configuration file. Custom algorithms can be implemented and added into the framework by simply inserting function pointers into a set of function tables.

Section 4.2: Clustering Manager

The clustering manager is responsible for node admission, internode communication, and tracking the statistics of other nodes in the cluster. Furthermore, the manager must identify dead or unresponsive nodes and purge them from the cluster. If the problem node is the current leader of the cluster, called the coordinator, the clustering manager must initiate an election among remaining nodes in the system to agree upon a new leader. Although load balancing can be performed in a decentralized fashion, ICM provides the ability to maintain a coordinator if the higher level algorithms require it.

Table 1 depicts the API exposed by the clustering manager to other modules of ICM. In order to spawn network daemon threads that watch for cluster communication, *icm_init* must be called on node startup. In addition, the initialization function sets control parameters that influence connection timeouts, port numbers, backlogs, and memory usage thresholds. By setting a memory usage threshold, the user can limit the amount of memory available for use by ICM processes. Communication is initiated with a node chosen by the decision engine as the target for remote invocation, which can potentially be the local host, by *icm_start*. The clustering layer then calls the VM driver on the target node, informing it of the application to be run. When the job launches, the clustering manager informs the caller that the job began successfully via a callback. Similar to *icm_start*, *icm_migrate* relocates a currently running process from a source node to a target node. Greater detail about the implementation of the start and migrate operations can be found in the companion document that deals with KVM interaction.

Table 1: API exposed by clustering layer.

CLUSTERING MANAGER API	
<i>icm_init</i>	Creates daemon threads to handle network communication. Initializes configurable control parameters.
<i>icm_start</i>	Launch specified application on target node.
<i>icm_migrate</i>	Relocate task from source node to target node.
<i>icm_join</i>	Joins existing ICM cluster.
<i>icm_leave</i>	Leaves ICM cluster.
<i>icm_updatestats</i>	Explicitly inform others of your system state.
<i>icm_requeststats</i>	Explicitly request system state of a specific node.
<i>icm_querystatus</i>	Inquires whether a target node is alive and responsive.

A node joins ICM by calling *icm_join*. On startup, a new ICM node initializes its own internal data structures for tracking cluster state, and then attempts to contact the coordinator specified in the configuration file. When the coordinator receives a join request from a new node, it replies with a unique identifier, the ICM id number, and the number of nodes that already belong to the system. The coordinator then sends detailed information about each node in the cluster to the newly joined node. In addition, messages are sent to each existing member of the cluster informing them of the details of the new node. Figure 2 depicts the flow of these messages. In practice, some of these messages can be piggybacked to minimize network traffic; however, for clarity, they are presented as individual transmissions in this description.

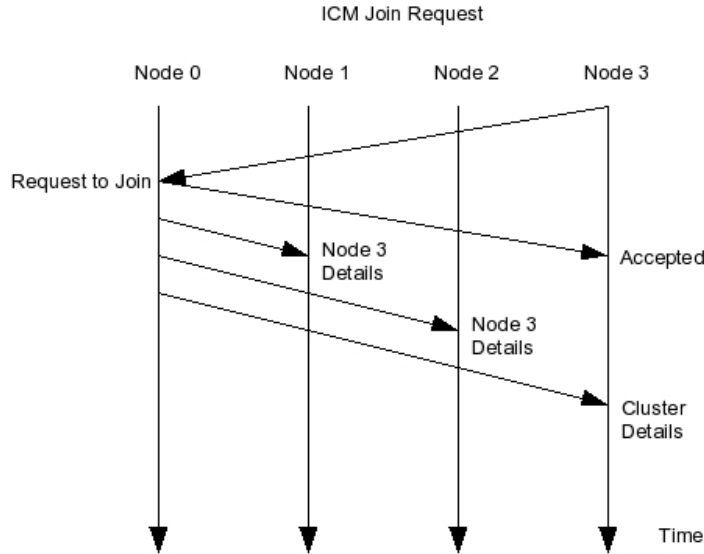


Figure 2: Logical sequence of messages when a node joins the cluster. Node 0 is the coordinator, and Node 3 is the joining node.

Similar to *icm_join*, *icm_leave* informs the coordinator of the node's intent to leave the ICM cloud. Upon receiving a leave request, the coordinator instructs all other nodes in the cluster to remove the leaving node from their view of the cluster and then removes the node from its own data structures. The sequence of network transmissions is similar to that presented in *icm_join* and is therefore omitted from this text. *Icm_updatestats* and *icm_requeststats* allow a higher layer module to force the refreshing of host statistics within the cluster. *Icm_querystatus* pings a specific node to determine its current state.

Section 4.3: Load Balancing

In this prototype implementation, load balancing consists of two parts: the load balancer and the decision engine. The load balancer is responsible for deciding whether a preemptive migration should occur, and if so, which nodes should participate as the source and destination. The decision engine plays the role of determining when jobs should undergo remote execution. In both cases, we have constructed ICM such that the algorithms used for each task are fully

configurable and replaceable modules. For the prototype, we implement several algorithms for each module to demonstrate the extensibility of the system.

Currently, ICM supports two load balancing algorithms. The first technique, the random algorithm, locates over-utilized nodes in the system and migrates a job to a random victim node which is less loaded than the source node. The algorithm uses the notion of a victim buffer as it searches the local host's data structure representation of the cloud. Each victim node's id and load is recorded during a single pass of all ICM nodes. A random number selects a victim from all available victims, and a migration is initiated if all the configurable threshold conditions are met. This algorithm scales linearly on the order of the number of nodes in the cluster.

Alternatively, the minmax algorithm attempts to migrate jobs from the most heavily utilized node within the cloud to the most lightly loaded host. If the minimally loaded node's load is above a certain threshold, the algorithm concludes that all nodes are reasonably loaded, and no significant performance gain will arise from a preemptive migration. Like the random selection technique, this method scales linearly on order of the number of nodes in the cluster.

The decision engine uses node state information to perform nonpreemptive load balancing. The decision engine presently supports two modes of operation: FTNM and LLS. FTNM, or "first that's not me," simply picks the first available node that is not the current node for remote invocation of a job. This algorithm does not require searching any data structures and can be implemented in constant time regardless of the size of the cluster. LLS, or "light load set," builds a set of all nodes which fall beneath a configurable light load threshold and picks a target node from this set at random. This technique scales linearly on the order of nodes in the cluster.

Section 4.4: Shell

The shell mimics a standard UNIX shell to allow command line input of commands to ICM. The ICM shell reserves *quit*, *help*, *info*, and *version* for ICM management commands: *quit* causes the node to withdraw from the cluster and shutdown the system; *help* displays information on using ICM; *info* displays node statistics such as the number of currently running ICM jobs, number of nodes in the cloud, and load indexes; and *version* presents the current version of ICM the node is running. All other commands are passed into the clustering layer to be executed within the resource cloud transparently to the user. The ICM shell supports general shell features including backspacing, inline editing, and a command history of configurable length.

Section 4.5: Statistics Daemon

Load balancing and remote invocation decisions are made based on statistics collected locally at each node. To collect the necessary statistics, a statistics daemon, together with the information dissemination mechanism described in the next section, implement the information policy of the system. The daemon runs once every second, reading appropriate *proc* file system entries. Currently, the daemon collects the one-minute load average, provided by */proc/loadavg*; the total available and free memory, provided by */proc/meminfo*; and an assortment of information about CPU usage, provided by */proc/stat*. While the CPU usage statistics are not utilized by any of our current algorithms, they are available for use in future implementations. The collection of the load average statistics is based on the *mpstat* utility source code. Statistics are delivered to remote nodes by means discussed in the next section.

Section 4.6: Election Daemon

Aspects of ICM, such as load balancing, rely on the existence of a single node designated as the system's coordinator. At any time, the coordinator may leave the cluster, either through a crash or via an interface exposed by the clustering manager. In the latter case, the coordinator may select a node from its list of known nodes and promote it as the new coordinator. The new coordinator sends a message to all other nodes in the cluster, informing them of the change. The transition is visible only to the clustering layer and all other nodes accept the new coordinator transparently.

In the event that a coordinator should leave the cluster without successfully promoting a new coordinator, most likely due to a system crash, the remaining nodes participate in an election. The result of this election is an agreement on a new coordinator. ICM uses a slightly modified version of the bully algorithm [14] for coordinator election.

Upon discovering the death of a coordinator, any node may start the bully algorithm election process by sending an *election* message to all nodes with a node id lower than its own. The node then waits for an *election reply*. If it does not receive one within a certain time, in our case 50 s, the node declares itself as the new coordinator by sending a *coordinator* message.

Upon receiving an *election* message, if the id of the receiving node is higher than its own, the node sends an *election reply*. Otherwise, the node ignores the message. The node then initiates its own election by sending an *election* message to all nodes with a lower id.

Upon receiving an *election reply* message, if the message has a lower node id than its own, the node waits for a *coordinator* message. Otherwise the node ignores the message. If the node does not receive a coordinator message within a certain time, in our case 50 s, it reinitiates an election.

Upon receiving a *coordinator* message, if the message came from a node with a lower id, the node accepts the new coordinator. Otherwise, the node starts a new election. Alternatively, the node could ignore *coordinator* messages with higher ids, but this would not support promotion.

In order to detect changes in the cluster, each node sends a heartbeat message to the coordinator once every heartbeat interval, in this case, every 5 s. If the coordinator does not receive a heartbeat from a node within a heartbeat timeout interval, in our case 2 min, the coordinator declares the node dead and informs all other nodes in the cluster. Conversely, the coordinator heartbeats to all other nodes in the cluster. To limit the amount of communication overhead, the coordinator heartbeats to a subset of the nodes during each heartbeat interval. We refer to the period during which all nodes receive a heartbeat from the coordinator as the *heartbeat epoch*.

To keep node state information up to date throughout ICM, node statistics information is piggybacked onto the heartbeat messages. Each node sends its own statistics to the coordinator during each heartbeat interval. Conversely, in order to limit the size of heartbeat messages, the coordinator sends the statistics for a subset of all nodes during each heartbeat epoch. We call the time for statistics for the entire set of nodes to be distributed to the entire cluster the *information epoch*. Obviously, depending on configuration parameters and the number of nodes in the cluster, the information epoch may be long, resulting in noncoordinator nodes not necessarily having up-to-date statistics on all other nodes. The coordinator, on the other hand, does have the most recent statistics, and is therefore responsible for load balancing.

Section 4.7: VM Manager

The VM manager consists of a manager thread per virtual machine instance in conjunction with a modified version of the virtual machine. Modifications to the virtual machine are required in order to enable signaling to and from the manager thread. Signaling is implemented by means of two one-directional message queues. An additional thread runs within the virtual machine, waiting for messages from the manager thread, and in response invokes appropriate calls to the VM's API. The manager thread, on the other hand, signals the virtual machine and waits for response messages only at precise points during the execution sequence.

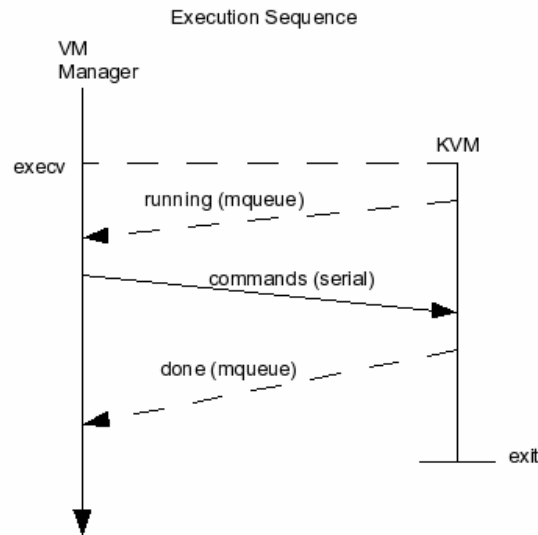


Figure 3: The execution sequence of a VM manager thread.

When the shell interprets a command other than one reserved for ICM control, it spawns a new manager thread. The execution flow of this thread, as well as the VM instance, is depicted in Figure 3. This thread creates the message queues, prepares arguments for the virtual machine, adds the job to the host's job list, and forks a new process which in turn calls *execv* to launch a new virtual machine instance. In our case, the manager thread starts KVM with arguments

specifying the virtual hard disk image and two subdirectories of the NFS file system. One of the NFS subdirectories is used as a read-only VVFAT disk containing executables and input files while the other is used as a read-write VVFAT disk serving as the destination for output files. Additional arguments instruct KVM to start in nongraphical mode, specify the size of memory to allocate for the instance, and load a premade state file. Loading from the state file saves boot-up time and brings KVM to a point where the Linux shell is waiting on command input. The manager thread then waits for a message from the virtual machine.

After initialization, the virtual machine signals to the manager thread that it is ready to accept commands. Commands are sent to the shell, running within the virtual machine, via an emulated serial port. In nongraphical mode, KVM reads input from *stdin* and redirects it to the serial port. We modified Linux configuration files within the virtual machine to provide a serial terminal and set up a unidirectional pipe such that writing to a stream from the manager thread is visible to the virtual machine instance's shell. The manager thread issues commands to mount the two VVFAT drives and execute the user command. At this point, the manager thread marks the job as ready to migrate to enable selection by the transfer policy. The manager then issues commands to unmount the VVFAT drives and gracefully shut down when execution of the command completes. The manager thread then waits on the pid of the *forked* process to complete either by a successful migration or by completion of the command.

At any point after the manager thread marks the job as ready to migrate, the virtual machine may be signaled to migrate to another node. Details of this process are discussed in the next section.

Section 4.5: Migration Daemon

Figure 4 demonstrates the flow of execution, during migration on the coordinator source node, and destination node. Event A corresponds to the spawning of a new virtual machine by the manager thread, as discussed in the previous section. Event B corresponds to the virtual machine signaling the host that it is ready to run. After sending the virtual machine the commands to execute, the host marks the job as ready to migrate. This does not mean that the job has to be migrated, but rather that, at this point, the job is ready to do so.

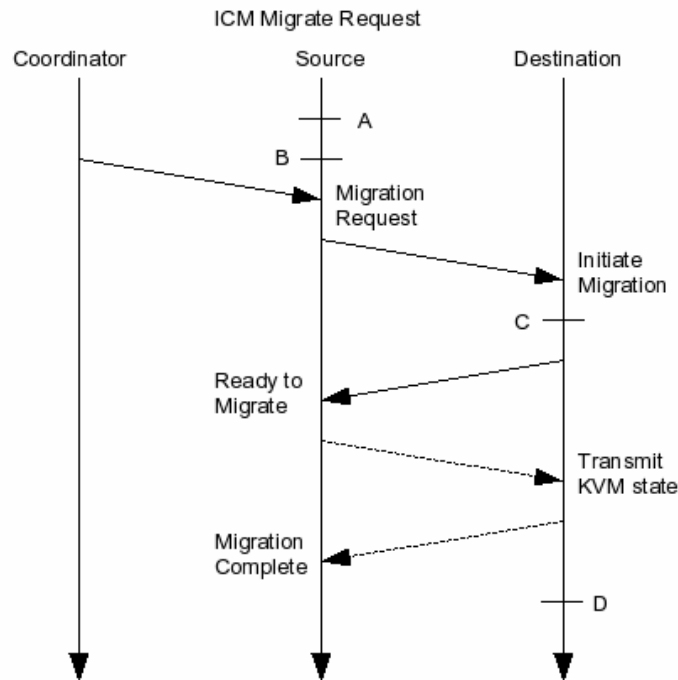


Figure 4: Major events and flow of messages between the coordinator as well as the source and destination nodes during migration.

Migration starts when the load balancer, running on the coordinator, decides that a source node is overly loaded and that one of its jobs should be moved to a destination node. The coordinator sends a *migration request* to the source node, informing it of the chosen destination node as per the location policy. The source node selects a job which is ready to be migrated and sends an *initiate migration* message to the destination node, informing it to prepare a virtual

machine instance for incoming migration. The message includes the NFS file system subdirectory to which any output should be written.

Event C corresponds to the virtual machine signaling the destination node that it is ready for an incoming migration. The destination node then replies to the source node with a *ready to migrate* message. The contents of this message include a port on which the virtual machine listens for an incoming migration.

Upon receipt of the *ready to migrate* message, the source node signals the virtual machine to stop execution and start an outgoing migration. The source virtual machine transmits its state and waits for an acknowledgement from the destination virtual machine. When the migration completes successfully, the source virtual machine signals the source host that it has finished migrating, and the host in turn signals the virtual machine to terminate. The destination virtual machine signals the destination host that it has finished migration and starts execution without the need for signaling. At this point the destination node again marks the job as ready to migrate within its own job list.

To prevent migration of a job back and forth between nodes without making any forward progress, jobs are added at the tail of a node's job queue and chosen for migration from the head.

Section 5: Evaluation Methodology

This section is divided into three sections. First, we discuss the setup of our test cluster. We follow with an overview of the specifics of KVM setup. Finally, we motivate the use of *bzip2* as our benchmark.

Section 5.1: The Test Cluster

For testing purposes, we have set up a small cluster consisting of four identical nodes. Each is a Dell Precision 390 workstation with a 2.66-GHz Intel Core 2 Quad processor with a 1066-MHz front side bus, a 4-MB L2 cache, and 4 GB of 533-MHz DDR2 memory. The workstations have onboard 1-Gbps network interface cards and connect to the cluster through our Netgear FWG114P 54-Mbps router. Each node is running Fedora Core 6 Linux with a 2.6.22.14-72.fc6 64-bit kernel. Figure 5 illustrates the setup of the test cluster.

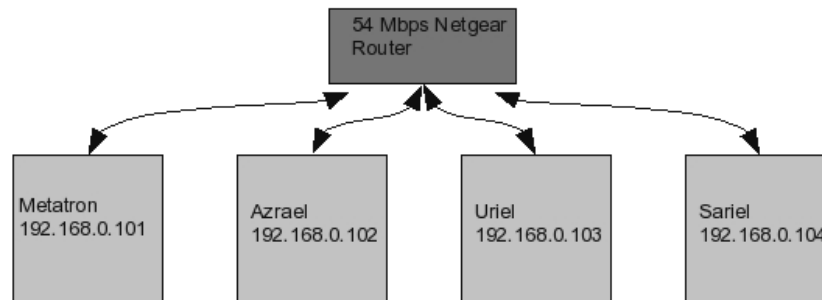


Figure 5: The four-node test cluster.

We have named the four nodes Metatron, Azrael, Uriel, and Sariel after angels from Judeo-Christian-Muslim mythology. In all experiments, Metatron serves as both the cluster coordinator and the network file system (NFS) server. We set aside a separate hard disk partition for this purpose. In order to provide some level of consistency, each node, including Metatron, mounts the NFS drive on a local directory. Metatron still has a slight advantage, in terms of NFS

overhead, over the remaining nodes since communication is performed over the loop-back interface. An alternative and preferable approach would have been to set aside an additional workstation to act as the NFS server. We were unable to take this approach due to lack of resources.

Section 5.2: KVM Setup

As previously mentioned, we have modified KVM to run within the ICM framework. The latest version of KVM available to us at the time of implementation was KVM-62. Coincidentally, this was the first version to fully support live migration. Other than our modifications, KVM is configured and compiled with the default options. Kernel support is loaded as a module, rather than being compiled into Linux, allowing ICM to run on a completely unmodified kernel. The KVM modules require hardware virtualization to be enabled on the host workstations.

KVM relies on a modified version of QEMU for emulation. We emulate a 2.66-GHz 64-bit x86 machine with a 2-MB L2 cache. QEMU instances are started with three hard disk images. The first is a 10-GB qcow disk image created with the *qemu-img* utility. We store a qcow image locally on each workstation to avoid NFS overhead. We have loaded the qcow images with Fedora Core 7 Linux running a 2.6.21-1.3194.fc7 kernel. The other images are subdirectories of the NFS file system encapsulated by QEMU's VVFAT layer.

Section 5.3: The Benchmark

We have selected the *bzip2* utility to serve as our performance benchmark. A modified version of the utility is part of the SPECint 2000 benchmark suite, making it a well established

benchmark of CPU performance. We expect that *bzip2* is representative of the type of jobs susceptible to execution within our framework. As a compression utility, *bzip2* is both computationally and I/O intensive, with the potential of stressing our system. Furthermore, *bzip2* does not require any interprocess communication or user interaction, features not currently supported by our prototype.

Each iteration of the benchmark consists of compression of three files—a TIFF image, an executable, and a source TAR archive—which are a good approximation of common workloads. We execute the benchmark in increments of 10 and 50 iterations, which we refer to as short jobs and long jobs, respectively. When executing multiple jobs, we space their submission by 10 s in order to simulate a reasonable average job arrival interval.

Section 6: Performance

In this section we analyze the performance of ICM when compared to that of a single workstation. In the first section we present some micro-benchmarks, taken on the host, as a reference for the reader. We follow in the second section with some micro-benchmarks measuring the overhead associated with ICM. In the remaining sections, we compare the performance of ICM with that of the host. Additionally, Section 6.6 compares the performance of various load balancing and remote invocation algorithms within the ICM framework.

Section 6.1: Host Microbenchmarks

Table 2 represents, for the sake of comparison, the overheads associated with launching a job on a host node, without ICM. Execution in the context of ICM involves spawning a management thread per virtual machine invocation, *forking* a process within the thread, and making the *execv* system call to launch QEMU. Thread creation takes on average 34 μ s, *fork* takes 95 μ s, and *execv* takes 510 μ s from the time it is called until the process being executed starts running. Together, these operations take on average 639 μ s.

Table 2: Thread creation, *fork*, and *execv* overhead on the host (in μ s).

Test	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
Thread Creation	38	30	37	30	37	34
Fork	106	90	96	87	95	95
Execv	567	560	549	327	548	510
Create Thread + Fork	144	120	133	117	132	129
Create Thread + Fork + Execv	711	680	682	444	680	639

We measure the time to create a thread by calling *gettimeofday* right before a call to *pthread_create* and again as soon as the thread starts running, then taking the difference between the two values. We measure *fork* in a similar manner, calling *gettimeofday* right before the call

and again when the *forked* process begins execution. Finally, we measure *execv* by calling *gettimeofday* right before the call and executing an application which again measures the time as soon as it starts running.

Section 6.2: ICM Microbenchmarks

Table 3 represents the overhead associated with ICM and the virtual machine. We measure an average of 1.3898 s from the time a user presses RETURN after entering a command in the shell, until the time that command starts running within the virtual machine. This includes the overhead of KVM signaling to the manager thread via a message queue that it has started running. Some additional overhead is associated with invoking a command remotely. The approximately 60-ms overhead can be attributed to creation of a thread on the local node to handle the memory copying of data between the child thread and its parent, as well as network communication cost. We see an average total of 1.4415 s from the time a user enters a command in the shell on the local node, until the time the command starts running within a virtual machine on a remote node. Once again, this cost includes signaling overhead.

Table 3: ICM and KVM overhead (in seconds).

Test	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
ICM+KVM Local Execution Overhead	1.3857	1.3840	1.3809	1.3845	1.4138	1.3898
ICM+KVM Remote Execution Overhead	1.4421	1.4421	1.4418	1.4405	1.4408	1.4415
KVM Shutdown Time	24.1393	25.0428	24.1266	24.0207	24.1043	24.2867
KVM Runtime Without Command	26.2302	25.6668	25.5487	25.5961	25.5214	25.7126
Time Per Migration	10.4353	10.4984	10.4304	10.4338	10.4472	10.4490

In order to prevent corruption of the virtual hard disk image, each virtual machine instance must gracefully shut down after executing a command. In KVM, the shutdown sequence

takes approximately 24.2867 s. While the job actually completes before the shutdown sequence completes, the user should not access any data produced until such time.

The cost of launching KVM and immediately starting the shutdown sequence is on average 25.7126 s. This includes writing to a file which indicates to the user that the job has finished. As expected, this cost is approximately equal to the sum of the local execution overhead and shutdown sequence cost.

Finally, we measure the average time to migrate a job from one node to another at approximately 10.4490 s. We make this measurement by calling *gettimeofday* just before signaling KVM to start a migration and again when KVM signals that it started running on the remote node. The cost of transmitting the state of the virtual machine over the network comprises most of this overhead; however, our implementation incurs some additional overhead of signaling through message queues.

Section 6.3: Single Short Job

We present the time to complete a single short job both on a host node without ICM, and through a single ICM node in Table 4. The nearly 100% overhead can be attributed almost entirely to the overhead of running an empty command discussed in Section 6.2. The remaining 2.3658 s, or approximately 7.7% of pure execution time, is the cost of executing in an emulated environment. We will demonstrate that the majority of the overhead can be amortized with increasing job lengths. After subtracting the total ICM and KVM execution overhead, an 8.3% overhead remains. This overhead is completely attributed to execution within the virtual machine. While the 8.3% overhead is not amortizable, it is much better than the anticipated 15 to 20%.

Table 4: Time to complete a single short job on the host and through ICM (in s).

Test	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
host.run-bzip2.1x10	28.4548	28.4151	28.4325	28.4297	28.3918	28.4248
icm.run-bzip2.1x10	56.4811	56.5047	56.5136	56.5067	56.5101	56.5032

Section 6.4: Single Long Job

Table 5 shows the time taken to complete a long job on a host without ICM as well as on a single ICM node. As expected, the ICM overhead is better amortized with an increase in job execution time. With the long job, we see an overhead of approximately 22.3%. Long jobs are approximately five times longer than short jobs and the overhead decreases inversely. Pure execution time, in this case, incurs an overhead of approximately 4.0%. We suspect that this overhead is lower than in the short job case since a larger number of iterations of the *bzip2* benchmark warm up memory and the processor’s caches. Additionally, some performance improvement may be seen due to the semantics KVM uses when writing to a VVFAT partition. KVM buffers writes and may not flush until multiple iterations are completed, decreasing the cost of writing over NFS.

Table 5: Time to complete a single long job on the NFS server, another node, and through ICM (in s).

Test	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
host.run-bzip2.1x50	141.7895	141.3846	141.4938	141.4587	141.5956	141.5444
icm.run-bzip2.1x50	174.5550	174.4709	173.4846	170.8619	172.4530	173.1651

Section 6.5: Ten Short Jobs

Table 6 shows the completion times (relative to the start of the first job) of 10 jobs started at 10-s intervals on a four-node ICM cluster. The jobs are listed in ascending order of completion time. While we will analyze the performance of the system later in this section, we would like to

mention a few interesting observations about their execution. Since the jobs are relatively short, no migration occurred during their execution. We used the LLS algorithm for remote invocation selection.

The first four jobs always finished on node 0, the coordinator and originator of jobs. Due to the semantics of the LLS algorithm, node 0 was considered to be lightly loaded until its 1-min load average increased sometime after the invocation of these four jobs. While they finished before they could be migrated, further jobs were invoked on remote nodes before the 1-min load average decreased to a lightly loaded level. The table clearly shows that the distribution of jobs across the cluster was relatively even.

Table 6: Time to complete 10 short jobs, started at 10-s intervals, on a four-node ICM cluster. The table also includes the node numbers of the nodes on which the jobs finished.

Trial 1		Trial 2		Trial 3		Trial 4		Trial 5	
Finishing Node	Finish Time	Finishing Node	Finish Time	Finishing Node	Finish Time	Finishing Node	Finish Time	Finishing Node	Finish Time
0	56.5046	0	56.4390	0	57.4056	0	57.4769	0	55.5226
0	68.4713	0	66.5191	0	67.4303	0	66.4318	0	67.5347
0	78.5192	0	76.5784	0	77.5588	0	76.4662	0	78.5429
0	87.5211	0	87.5638	0	85.6142	0	87.4986	0	87.7573
1	99.2653	3	96.5983	3	96.6850	1	96.5081	1	95.5744
1	105.5459	3	106.5617	3	107.6488	1	106.5222	3	106.6791
3	116.6255	1	117.5120	3	117.6741	2	116.4305	1	116.5841
2	126.5112	3	125.6550	2	126.4518	3	126.5776	2	126.4672
2	136.4702	1	136.5040	2	136.4459	1	136.5983	3	135.5765
3	146.5777	1	146.5416	1	146.5434	2	146.5373	2	145.4827

Table A.7: Time to complete 10 short jobs on a single host, a single-node ICM cluster, and a four-node ICM cluster.

Test	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
host.run-bzip2.10x10	121.3600	121.0231	120.6279	119.9011	120.2022	120.6229
icm.run-bzip2.10x10.1_node	149.6349	148.7241	149.6529	149.5765	149.7650	149.4707
icm.run-bzip2.10x10.4_nodes	146.5777	146.5416	146.5434	146.5373	145.4827	146.3365

Table 7 compares the performance of a single host without ICM, a single-node ICM cluster, and a four-node ICM cluster. Table 6 describes the latter of these in more detail. As expected, due to the length of jobs, the single host outperforms both the single-node ICM cluster and the four-node cluster, consistently with the ICM overheads discussed in Section 6.2.

The performance gain of the four-node cluster is marginal over that of the single-node cluster. While seemingly surprising at first, this once again can be attributed to the short execution time of the jobs. A single ICM node can run four instances of the virtual machine with little overhead. Each virtual machine can run on one of the four cores, incurring slight overhead only when performing I/O. By the time a fifth job is invoked, the first virtual machine instance is in the shutdown phase, so the fifth can take its processor. By the time the sixth is invoked, the second is about ready to relinquish its processor, and so on. Once again we can clearly see that short jobs are not very susceptible to performance gains on ICM.

Section 6.6: Ten Long Jobs

Table A.8 shows the time to complete 10 long jobs (relative to the start of the first), on a four-node ICM cluster using the random load balancing algorithm and LLS remote invocation algorithm. The jobs are listed in ascending order of completion time. While the jobs are still too short for the cluster to reach steady state, the table demonstrates a much more even distribution of jobs. In the first case we see the optimal 3-3-2-2 distribution, while three of the remaining cases show the next best 4-2-2-2 distribution. The remaining 4-4-2 case can only be attributed to the random decision making process of the system, as well as to the fact that the system had not yet reached steady state.

Table 9 demonstrates the results of the previous experiment repeated using the minmax load balancing algorithm and LLS remote invocation. We see slight improvement in performance. The system is likely to reach the optimal 3-3-2-2 job distribution more quickly, due to the semantics of the algorithm, resulting in the measured improvement.

Table 8: Time to complete 10 long jobs, started at 10-s intervals, on a four-node ICM cluster using random load balancing and LLS. The table also includes the node numbers of the nodes on which the jobs finished.

Trial 1		Trial 2		Trial 3		Trial 4		Trial 5	
Finishing Node	Finish Time	Finishing Node	Finish Time	Finishing Node	Finish Time	Finishing Node	Finish Time	Finishing Node	Finish Time
1	195.6967	2	206.1252	2	194.6577	3	198.5466	0	184.4579
3	210.9320	0	212.5552	0	213.6098	2	212.6612	3	206.9193
0	211.6796	3	218.1474	3	227.7388	1	222.8396	3	216.7797
3	220.9032	1	223.8485	0	234.9815	1	225.6319	2	237.9546
1	237.5183	3	240.9415	0	241.0274	1	237.0258	2	245.5372
0	237.9010	3	248.0569	3	243.7379	2	247.7592	0	248.8430
2	243.4838	2	263.1261	3	247.9618	2	271.8927	2	252.8650
2	255.5247	3	265.1444	2	274.4503	3	272.0360	3	258.7804
2	257.7086	0	274.7310	0	287.1881	2	273.5873	0	268.7793
1	270.9315	3	283.9868	2	308.3691	1	282.9697	3	289.0419

Table 9: Time to complete 10 long jobs, started at 10-s intervals, on a four-node ICM cluster using minmax load balancing and LLS. The table also includes the node numbers of the nodes on which the jobs finished.

Trial 1		Trial 2		Trial 3		Trial 4		Trial 5	
Finishing Node	Finish Time	Finishing Node	Finish Time	Finishing Node	Finish Time	Finishing Node	Finish Time	Finishing Node	Finish Time
0	182.4867	0	184.4864	2	195.6999	0	182.5347	1	195.8222
3	204.9044	0	204.4859	2	208.7304	1	206.8791	2	209.6926
3	219.0804	2	207.6904	0	211.5695	1	217.8814	0	210.7101
2	231.0014	0	213.5549	1	219.9223	3	224.7820	1	216.8589
2	233.5601	1	221.6317	1	237.7328	3	229.1229	3	217.7878
2	239.8171	3	230.7756	2	240.4935	3	236.0111	2	236.7002
2	256.7248	2	238.5327	0	247.7487	1	240.7482	1	237.7208
3	257.8091	3	240.7758	3	257.8380	0	278.7258	3	250.8320
1	268.7362	2	258.4790	3	269.8664	0	278.8164	3	256.8521
1	278.7449	1	261.6751	1	275.8141	0	287.8052	2	271.2695

Table 10 repeats the previous experiment with minmax load balancing and the naïve FTNM remote invocation algorithm. Here we see a substantial performance decrease. As the name, “first that’s not me,” implies, when a job is invoked on node 0, it is automatically remotely invoked on the first node in the list of all nodes. Since the list never changes, this is always the same node without regard for the load. The chosen node becomes so loaded that it is slow to respond to migration requests. In some cases migration target nodes appear to time-out waiting for migration packets, exit, and in turn cause the jobs on the heavily loaded source node to never migrate, while still incurring the migration overhead costs.

Table A.10: Time to complete 10 long jobs, started at 10-s intervals, on a four-node ICM cluster using minmax load balancing and FTNM. The table also includes the node numbers of the nodes on which the jobs finished.

Trial 1		Trial 2		Trial 3		Trial 4		Trial 5	
Finishing Node	Finish Time	Finishing Node	Finish Time	Finishing Node	Finish Time	Finishing Node	Finish Time	Finishing Node	Finish Time
2	201.6485	2	195.1509	2	210.4108	2	214.0232	2	214.1630
2	217.7242	1	217.9734	1	302.9704	2	289.2232	1	269.1209
0	244.6698	1	233.9645	1	304.4793	1	305.8320	0	279.0939
0	257.5890	1	271.1625	2	313.7671	1	311.9519	2	294.3652
3	264.1137	3	287.9115	0	318.8339	0	329.1756	2	315.5533
2	299.2173	1	291.0940	2	343.5242	0	333.6570	0	334.2537
2	305.2073	2	295.9799	2	346.1318	1	354.4328	1	367.8138
1	305.7552	3	295.9861	1	359.8994	3	364.1588	2	373.5101
3	310.9788	0	303.0730	2	371.7935	0	394.3236	2	377.1537
1	334.7063	1	310.8853	0	413.9282	1	422.4223	1	385.2467

Table 11 shows the execution time of 10 long jobs on a single host without ICM, single-node ICM, and multinode ICM, and summarizes the results of the previous three experiments. Execution of 10 jobs on the host takes on average 470.4347 s. Single-node ICM does slightly worse. This can be attributed to ICM overhead. The two LLS experiments show a

39% and 42% performance gain for random and minmax load balancing algorithms, respectively. Even the naïve FTNM experiment sees some performance gain, although only 21%.

Table 2: Time to complete 10 long jobs, started at 10-s intervals, on a single host, a one-node ICM cluster, and a four-node ICM cluster. Three combinations of load balancing and remote invocation algorithms are presented.

Test	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
host.run-bzip2.10x50	481.6638	468.2212	471.5711	463.1089	467.6086	470.4347
icm.run-bzip2.10x50.1_node	498.6787	530.0874	508.0458	540.4312	513.0114	518.0509
icm.run-bzip2.10x50.4_nodes.rand-lls	270.9315	283.9868	308.3691	282.9697	289.0419	287.0598
icm.run-bzip2.10x50.4_nodes.min_max.lls	278.7449	261.6751	275.8141	287.8052	271.2695	275.0617
icm.run-bzip2.10x50.4_nodes.min_max.ftnm	334.7063	310.8853	413.9282	422.4223	385.2467	373.4377

The results here are a little disappointing because we hoped for about a 70% performance increase with the minmax and LLS algorithms. The discrepancy can be attributed mainly to the cost of migration. We expect, however, that as the length of jobs increases the system will reach steady state and cease migration, amortizing its cost over execution time.

Section 7: Conclusions and Future Work

We start this section with a summary of our results in briefly discussing their implications. We follow by listing some lessons learned from the project. Finally, we discuss some potential directions for future work on the Illinois Cluster Manager.

Section 7.1: Conclusions

In summary, our performance results have several implications. The performance differences between minmax and random load balancing algorithms are negligible for the sample workload we used. Minmax exhibits slightly better behavior because it reaches a stable state more rapidly than random migrations. Moreover, the significant overhead of ICM limits its usefulness when running short-lived jobs. When running longer tasks, the startup and shutdown times are amortized over the life of the process. Although short jobs do not directly benefit from being run in the ICM framework, they indirectly benefit from the migration of long jobs away from the system.

Section 7.2: Lessons Learned

Overall, ICM has provided us with insight into the process of designing and implementing a sophisticated distributed application that incorporates a variety of open source software. For future projects, we suggest following a more formal process of outlining the exact goals of the endeavor and designing a set of performance and functionality requirements that the implementation must meet. Frequent deadlines are helpful in maintaining steady progress. To facilitate code integration, we recommend agreeing upon a standard API for each major software

component. Additionally, we frequently found ourselves overextending the scope of the project beyond what we could feasibly accomplish in the project time frame. We advise remaining faithful to the goals established at the outset of development until all original objectives have been accomplished.

Section 7.3: Future Work

In our workload assumption, we have assumed that all processes are CPU-bound and perform no interprocess communication. Going forward, we could expand our target workload to include interactive and networked applications. By extending the clustering layer, we could implement a version of mobile IP to allow jobs to transparently roam the cluster while maintaining seamless network connectivity. In addition, we would like to enhance the sophistication of our load balancing algorithms to dynamically adjust for detected cluster conditions. Perhaps the balancer would become more aggressive when it detects an intense load on a small subset of the cloud, and more passive when the load approaches equality.

To address hardware outages, ICM can be augmented to support periodic check pointing. As a side effect of virtual machine migration, we have the ability to capture and save the state of a job at an arbitrary point in time. Implementing check pointing would only involve minor adjustments to the current framework. We expect such a feature to face overhead on the order of that seen for migration. Furthermore, the virtualization layer of ICM could be redesigned with a lighter weight virtual machine than QEMU to reduce performance overhead.

Acknowledgments

Lukasz Lempart would like to make the following acknowledgements.

First and foremost, I would like to thank James Pike, my friend and partner in this project, for the time and effort he invested to ensure that we both finished on time. My sincere gratitude goes to my adviser, Professor Matthew Ian Frank, for helping shape this project, always being there to provide ideas and opinions, and providing us with the necessary equipment. Special thanks go to Professor Steven Lumetta for inspiring me to pursue this path of study. I would like to thank my parents, Ryszard and Bozena, and my younger sister, Anna, for always pushing me to reach my full potential. Special thanks go to my friends, Jimmy, Bill, Matt, Ryan, and Chris, who always believed in me. I would also like to acknowledge the open-source community, especially the KVM and QEMU developers without whom this project would not be possible, for providing quality free software. Last but far from least, I would like to thank Sharon for her support, encouragement, and love. Without her always pushing me to work harder, I would not have been able to finish this project on time.

References

- [1] D. Arredondo, M. Errecalde, F. Piccoli, M. Printista, R. Gallard, and S. Flores, "Load distribution and balancing support in a workstation-based distributed system," *SIGOPS Operating Systems*, rev. 31, no. 2, 1997, pp. 46-59.
- [2] M. Harchol-Balter and A. B. Downey, "Exploiting process lifetime distributions for dynamic load balancing," *ACM Transactions on Computing Systems*, vol. 15, no. 3, pp. 253-285, August 1997.
- [3] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor-a hunter of idle workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems*, 1988, pp. 104-111.
- [4] E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad, "Design and implementation of a distributed virtual machine for networked computers," in *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, 1999, pp. 202-216.
- [5] W. Huang, J. Liu, B. Abali, and D. K. Panda, "A case for high performance computing with virtual machines," in *Proceedings of the 20th Annual International Conference on Supercomputing*, 2006, pp. 125-134.
- [6] D. S. Milojević, F. Douglass, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Computing Surveys*, vol. 32, no. 3, pp. 241-299, September 2000.
- [7] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the USENIX Annual Technical Conference*, 2005, pp. 41-46.
- [8] Qumranet Corp., "KVM: Kernel-based Virtualization Driver," February 2008. [Online]. Available: http://www.qumranet.com/wp/kvm_wp.pdf.
- [9] A. Whitaker, M. Shaw, and S. D. Gribble, "Denali: Lightweight virtual machines for distributed and networked applications," in *Proceedings of the USENIX Annual Technical Conference*, 2002, pp. 195-209.
- [10] J. Casas, R. Konuru, S. W. Otto, R. Prouty, and J. Walpole, "Adaptive load migration systems for PVM," in *Proceedings Supercomputing*, 1994, pp. 390-399.
- [11] M.Q. Xu, "Effective metacomputing using LSF Multicluster," in *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2001, pp. 100-105.
- [12] T. Kunz, "The influence of different workload descriptions on a heuristic load balancing scheme," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 725-730, July 1991.

- [13] N.G. Shivaratri, P. Krueger, and M. Singhal, "Load distributing for locally distributed systems," *Computer*, vol. 25, no. 12, pp. 33-44, December 1992.
- [14] H. Garcia-Molina, "Elections in a distributed computing system," *IEEE Transactions on Computers*, vol. C-31, no. 1, pp. 48-59, January 1982.