



Technischer Bericht

HARPY: A Virtual Machine Based
Approach to High-Throughput
Cluster Computing

Christian Kauhaus Andreas Schäfer

Friedrich-Schiller-Universität Jena
Institut für Informatik
Lehrstuhl für Rechnerarchitektur
und -kommunikation

Hrsg.: Prof. Dr.-Ing. W. Erhard

FSU

Christian Kauhaus Andreas Schäfer

HARPY: A Virtual Machine Based Approach
to High-Throughput Cluster Computing

HARPY: A Virtual Machine Based Approach to High-Throughput Cluster Computing

Christian Kauhaus Andreas Schäfer

15th February 2005

Editor:
Prof. Dr.-Ing. Werner Erhard
Friedrich-Schiller-Universität Jena
Institut für Informatik
Ernst-Abbe-Platz 1-2
07743 Jena, Germany

ISSN 0949-3042

Contents

1	Introduction	9
2	Background and Related Work	11
2.1	Terminology	11
2.2	Opportunistic High-Throughput Computing	11
2.3	General Options to Client-Side Program Execution	13
2.4	Examples of High-Throughput Systems	16
2.4.1	Sun Grid Engine	16
2.4.2	Condor	18
2.5	Discussion	19
3	Architectural Overview of HARP	21
3.1	General Design Considerations	21
3.2	Components	22
3.2.1	Client	22
3.2.2	Master Execution Node	24
3.2.3	Configuration Server	26
3.3	Events in HARP	27
3.4	Summary	28
4	Client Architecture	29
4.1	Construction of the Linux VM	29
4.1.1	Running a Linux Virtual Machine	29
4.1.2	Integration of a Process Migration Facility	31
4.1.3	Network Setup	32
4.2	Installation, Configuration, and Update	34
4.2.1	Installation Procedure	34
4.2.2	Configuration Issues	35
5	Performance Evaluation	37
5.1	Processor Usage	37
5.2	Memory Access	38
5.3	Networking Overhead	41
5.4	Summary	44

Contents

6 Summary and Outlook 45

Bibliography 49

List of Figures

2.1	Core cluster, user workstations, and HARPY pool	12
2.2	Options to Client Program Execution	16
2.3	Architecture of Condor	18
3.1	General architecture	23
5.1	Results of the POV-Ray benchmark on native Linux	38
5.2	Results of the POV-Ray benchmark on coLinux	38
5.3	Memory access times on Linux	39
5.4	Memory access times on Windows	40
5.5	Memory access times on coLinux	41
5.6	Networking data flow through coLinux	42
5.7	Timing diagram for packets	43

1 Introduction

With standard desktop computers becoming more and more powerful, the amount of unused computing power is constantly rising. Over the last years, many systems have been proposed to harness these unused resources. These pieces of middleware are generally characterised as *high-throughput* systems, providing facilities for the convenient handling of thousands of independent jobs, and as *opportunistic* systems, trying to make use of computers while they are running otherwise idle [TTL04]. In many typical environments there is a Windows PC on each desk, but larger calculations run on a central Linux cluster or SMP compute server. In particular, the Beowulf concept [SSB⁺95] has got enormous popularity. Although both cluster and Windows PCs share the same hardware architecture, they use different operating systems and hence different programming models. While current opportunistic high-throughput systems are running well on several platforms, a single application is usually tied to the one platform it has been written for. Designing and implementing multi-platform programs is often not worth the effort, since it involves an additional layer of complexity: there are sometimes just too many details one must take into account. So even when there is a high-throughput middleware running on several platforms, developing multi-platform applications which utilise most of the computing power of a typical heterogeneous environment remains challenging.

As an alternative to multi-platform application development, we present a new approach combining several emerging technologies. The aim of the HARPY project is to bring two important platforms together: applications may use the resources of both a Linux cluster and idle Windows PCs. To achieve this goal, a virtual machine in form of a user-mode Linux kernel running on top of Windows is combined with a transparent process migration facility. HARPY causes applications running on a central cluster to be migrated to idle PCs. During all of this, the application is sandboxed in such a way that it does neither notice the migration nor any differences in its environment. So we try to get the combined computing power of two important platforms without modifications on either the application or the user PC.

Recently, interest has grown to use virtual machines in distributed systems.¹ Although the use of virtual machines is most often afflicted with per-

¹ A wide-known implementation is the sandbox for Java applets [MF99]. A mini-application (the applet) is transferred from a web server to the user's PC, where it is executed inside a virtual machine on behalf of the server. The execution is restricted in such a way that the

formance degradation, it provides an elegant solution to the problem that the environment a specific application was written for does not always match the environment the actual execution host is able to offer. In combination with high-throughput and opportunistic computing, which means to use as many machines as possible in order to handle large numbers of jobs, virtual machines provide a good way to reduce complexity. Our approach, the HARPY project, differs from other approaches in this respect that it does not create another abstract, purely virtual environment. We rather try to reproduce the known and well-understood environment of a dedicated, homogeneous cluster on a non-dedicated, distributed and (to some degree) heterogeneous network of workstations. The environment that is emulated this way happens to be an environment for which many applications are written and well tested. Therefore our approach also serves user groups who often do not write their applications from scratch, but tend to employ packaged, commercial software. Examples for this are solvers for common engineering problems or general mathematical suites.

Consequently, for HARPY the optimal deployment would be a departmental infrastructure which already has a compute server or a cluster for high-throughput style applications consisting of many independent jobs, but has problems with delays in application processing since the compute server/cluster cannot handle the work load fast enough. Instead of an upgrade of the compute server/cluster, with HARPY it is possible to use the processing power of the PCs usually found in every office. This way, we get a huge throughput gain without an expensive upgrade and without an elaborate application rewrite.

The name “HARPY” refers to the species of Harpy Eagles, the largest and most powerful living eagles. Harpy Eagles are found mainly in the rain forests of Middle and South America and are quite aggressive hunters. They grab everything they can, even smaller monkeys.

The rest of this paper is organised as follows. In Chapter 2, we briefly introduce the background of our project and discuss related work. Chapter 3 provides an overview of HARPY’s high-level design and describes its components. In Chapter 4, we perform a more detailed analysis for one of the core components: the client. Chapter 5 gives a preliminary performance study and Chapter 6 concludes with a summary.

applet cannot interfere with the local operating system. Other, more recent implementations (e. g. [WSG02]) supply much more capable virtual machines.

2 Background and Related Work

2.1 Terminology

In order to facilitate the understanding of the following description, a short list of terms is given first.

Cluster We use this term in a narrower sense to denote a central, tightly coupled group of computers usually located in a rack in the server room. To emphasise this, sometimes the term “core cluster” is used.

Workstation Computer on each user’s desk. These days, we find mostly PCs running Windows here.

HARPY Pool Whole HARPY system consisting of one or more core clusters and several workstations. For clarity, refer to Figure 2.1.

Distributed Resource Management (DRM) A System which controls the dissemination and execution of jobs over a set of computing resources.

User Application This term is exclusively used in the special sense that it is the program (or series thereof) started by the DRM and handled via HARPY.

Transparent Migration Generally speaking, migration is the transfer of a running process from one computer to another. In the case of transparent migration this is going to be done without any notice or interaction from the process itself.

Operating System (OS) This includes the OS kernel as well as standard libraries, utilities and system programs.

2.2 Opportunistic High-Throughput Computing

The domain of cluster computing is usually divided into three main fields of application: High-Performance (HPC), High-Throughput (HTC) and High-Availability computing (HAC) [Buy99]. The first aims to get the same work done faster, the second tries to get more work done in the same time, while the third concentrates on doing work more reliably. In this paper we focus

2 Background and Related Work

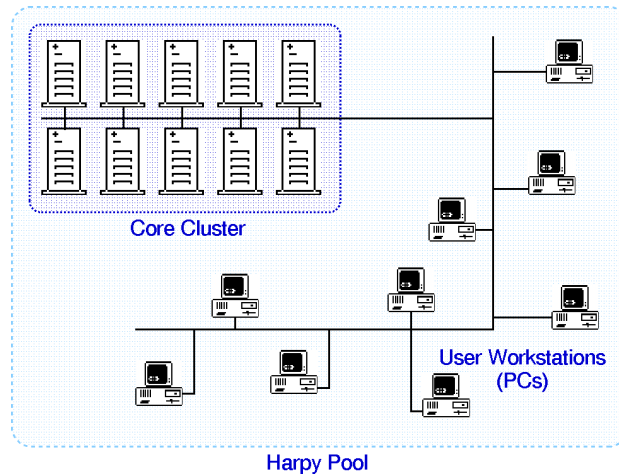


Figure 2.1: Core cluster, user workstations, and HARPY pool

on high-throughput computing, which is of use in science and engineering. For many tasks a parameter space has to be explored, thus needing *parameter studies* or *parameter sweeps*: the same application has to be run many times, each time with different input data. This may be needed to explore a full set of possible combinations (for example, in the simulation of electronic devices), or to perform an optimisation.¹ Generally speaking, in high-throughput computing it is not so important how fast an individual computation is completed, opposed to the need to complete as many computations as possible in a given time. It is therefore desirable to use the largest amount of available computing resources. This includes the utilisation of user workstations for those periods when they are left idle, leading to the concept of *Opportunistic High-Throughput Computing* [Wri01].

The key advantage of opportunistic high-throughput computing is that it enables the utilisation of computing resources which would otherwise be wasted. When a user workstation becomes idle, this condition is detected by a resource management software. Then, the execution of an application is transferred to the workstation. Of course, the migration is not for free. But even when we must accept a significant migration overhead and gain only little throughput, this is better than making no use of an idling workstation at all.

Of course there is a limit. Not every application is suited to be run on every hardware (e.g. due to memory constraints), and opportunistic high-throughput computing requires substantive higher management efforts than traditional approaches using dedicated hardware. Nevertheless, there are

¹ An example of such a *optimising parameter study* which invokes the same application many times is given in [FKK04].

many cases left where the use of opportunistic high-throughput computing leads to an overall gain.

2.3 General Options to Client-Side Program Execution

In opportunistic high-throughput computing programs are also executed on non-dedicated clients, which introduces additional management problems not present in purely dedicated environments. Many systems have been devised to accomplish this, both general purpose architectures and specially targeted developments. They differ widely in their feature list, in their job management, or in their user interface. But when it comes to the actual program execution on the client, most of them fall into three main categories:

1. Native execution
2. Sandboxing and interposition
3. Specialised client program.

All of these have advantages and disadvantages and are going to be explained in the following.

Native Execution In this most simplistic approach, an already existing distributed resource manager is extended. DRM systems have been used for a long time now on dedicated clusters, and usually consist of a job submission facility, a scheduler, and an execution agent. While the first two are centralised on special servers, an execution agent is placed on each execution node. DRM systems have originally been designed for use on dedicated computing resources, but some of them (for example, Portable Batch System [Ver00] or Sun Grid Engine [Sun02]) are flexible enough to monitor external activity on each node and act according to a specified scheduling policy. So in principle they allow for opportunistic high-throughput computing.

Configured this way, it is possible to let the DRM only start jobs on nodes running idle otherwise. Unfortunately, a number of problems exist using this approach. The application has to be specifically targeted to the execution platform and the whole runtime environment needs to be available on every node, including all necessary libraries etc. There is no protection between the DRM-started application and the local user's processes, so they may interfere. All nodes need to be under a consistent administrative control to guarantee unique user identities. The same mechanisms for data access (shared file systems, database connectivity etc.) have to be available on all nodes. Another serious problem is that there is only little support for process evacuation when

2 Background and Related Work

a local user becomes active again. This can be achieved using a checkpoint and restart service, but this again has to be deployed and configured on each execution node.

So the native execution model is only feasible when there are a lot of similar machines under a tight central control, which is the case in some office or classroom environments, and there are either only short-termed jobs to run or local resource use is predictable, so the scheduler may plan ahead accordingly. However, the biggest advantage of this model is that it implies hardly any performance loss: since there is no interposition layer whatsoever between the application and the host OS, it can use all local resources exactly like any other program.

Sandboxing and Interposition Systems in this category run applications in a restricted environment called *sandbox*, in contrast to the former category, where a native process start mechanism is utilised. The use of a sandbox together with the modification of system calls via an *interposition agent* [Jon93] gives the possibility of performing opportunistic high-throughput computing on a variety of client machines with no modifications on both the application program and the operating system.

A sandbox is an environment that shields the host computer from the programs running inside. This can be made possible by using already present operating system facilities (for example, the Solaris *proc* debugger trap [MM01]), by restricting the bytecode interpreter that certain languages need anyway (like Java [MF99]), by running user-mode versions of operating systems that behave like normal user processes to the host OS (for example, the User Mode Linux project [Dik01]), by emulating a virtual computer hardware in software (like VMwareTM), or by using partitioning mechanisms already built into certain platforms (mainly found on mainframe systems). Although these realisations differ widely from a technical perspective, they all provide an abstraction to the underlying platform which allows to monitor all accesses, and in turn to pass, block or alter them according to specified rules.

An interposition agent resides at this particular interface and translates system calls from inside the sandbox to appropriate calls to the outside. Interposition agents which are most useful for high-throughput computing forward system calls in such a way that the environment visible from inside the sandbox pretends to be a well known environment. If the emulated environment matches the environment an application was written for, the application is able to run unmodified in the sandbox. Difficulties arise from three sources here. First, the emulation of the original environment may not be perfect, thus restricting the application. Second, constructing the sandbox and an appropriate interposition agent may be hard on some platforms, restricting the choice of the host system and/or the interposition technique. Third, only a

certain fraction of the host computer's resources can be used inside the sandbox. This is particularly true for memory usage, but there are also restrictions on CPU, I/O, and network usage.

Interposition agents work generally by attaching to an interface between the application and its environment, intercepting calls, and altering or forwarding such calls as necessary. Although all interposition agents share this common principle, their implementations vary greatly in detail. A comparative study of several interposition techniques [TL03] reveals that there are eight general approaches, which have quite different properties with respect to restrictions and performance. In short, all known interposition techniques fall into two categories:

- *Internal* (library-level) techniques work by modifying the process space of an application in some way. Although they provide good performance, they tend to be very dependent on the libraries which are used by the application. The exact control and data flow through these libraries (most importantly the standard library `libc`) has to be known to effectively intercept all necessary calls. Since the internals tend to change between different library versions, this interposition technique is quite instable in face of library updates.
- *External* (kernel-level) techniques require modifications at the system call interface and are thus independent of libraries. Since there is only a small number of system calls with well defined call conventions, it is comparably easy to intercept them. The biggest problem here are either the performance loss that occurs due to an increased latency if the interposition agent itself resides in user space, or the difficulty to implement an interposition agent in kernel space.

Specialised Client Program Some quite successful high-throughput computing projects (for example SETI@home [KWA⁺01]) use a client program that is written specifically to perform its computation in a widely heterogeneous environment. Since it has already built in all precautions needed to execute on unknown resources as well as methods for data access, it is able to run on a wide range of clients without special configuration.

Of course, this requires the program to be specifically written for this purpose. All tasks usually performed by middleware or interposition layers need to be compiled into the application program. The program has also to be equipped with special methods for data access. Although there have been efforts to provide generalised toolkits for writing such programs², there is still the need to specially adapt a program for this purpose. This incurs a fairly

² For example, the “Bovine RC5” project of distributed.net [Dis04] has some generalised infrastructure for distributed cipher code breaking.

2 Background and Related Work

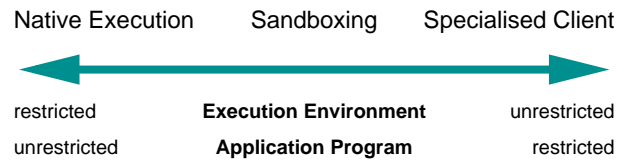


Figure 2.2: Options to Client Program Execution

expensive programming overhead in many cases and may only pay off when the client program is widely distributed.

To sum up, there are three main categories of program execution on the client. The “native execution” model puts severe restrictions on the clients, whereas the “specialised client program” model pushes most of the restrictions into the application program. The “sandboxing and interposition” model is somewhere in the middle (cf. Figure 2.2). This approach is quite promising, but also technically demanding.

2.4 Examples of High-Throughput Systems

In the following we will analyse two well-known systems that are in production use. The first one will be a typical installation of Sun Grid Engine, which belongs to the first category presented above (native execution). The second one will be a Condor pool, which may use an execution model belonging to the second category (sandboxing), although support for a native execution model is also available. We present no example from the third category in detail, since this is outside of the scope of this paper.

2.4.1 Sun Grid Engine

Sun Grid Engine (SGE) [Sun04] is a distributed resource management system which distinguishes three node types: the master node, submit nodes, and execution nodes. On every execution node, SGE runs two services: an *execution agent* which is capable to start and kill application processes, and a *load sensor*, which monitors the system state. These features allow SGE to be used for opportunistic high-throughput computing, although it has not been originally designed for this. The flow of a single job through the system is as follows:

- A user who wishes to run a job creates a job script, which consists of a shell script to conduct all necessary program invocations, enhanced with some descriptive control statements providing information for the

SGE scheduler. These control statements include required CPU and OS types, amount of available memory, etc.

- The job is submitted to the SGE scheduler. The scheduler matches the job's requirements with both the static (e. g. CPU type) and the dynamic (e. g. current CPU utilisation) information about the cluster that is inquired via the load sensors located on every execution node. In non-dedicated environments the scheduler usually considers only execution nodes which run idle, i. e. which have no notable local user activity.
- When a suitable resource for the job is found, the job script is started via the execution node's local execution agent. The job is usually run to completion, unless there is some sort of local checkpointing and restart mechanism available on the execution node. In this case the job may be stopped and optionally migrated away, triggered by some predefined node overload condition.
- When the job completes or aborts, the job's output is transferred back to the submit host.

There are several things to note here. *First*, the job as seen by the SGE is a shell script which may do all kinds of actions. This provides great flexibility, but prevents automatic file staging. Since it is impossible to analyse a shell script for all referenced files, all list of all accessed input and output files has to be given from the user. Due to the fact that this is cumbersome and does not reliably work for random access files, SGE usually relies on some sort of shared file system which makes all users' home directories transparently accessible on every execution node. *Second*, the user's application is executed under the same user ID which was used to submit the job. This means that the user IDs need to be synchronised between all nodes participating in a SGE cluster. This is not always simple to achieve, especially when it comes to heterogeneous installations. Although there have been efforts to provide some user ID translation, the issue is not completely resolved yet. *Third*, the user's job script is started by the execution agent just by the means of an ordinary `fork()` system call. Therefore we neither have a sandboxing mechanism nor some other form of protection or virtualisation. This means that the SGE application may crash the whole execution node. Furthermore, the application has to be compiled for exactly the same system environment it is executed on, since both libraries are dynamically linked and data access paths are accessed in a system dependent way. *Fourth*, when there is a possibility to checkpoint and optionally migrate jobs on the execution node (this has to be configured for every type of execution node), job migration is controlled by an overload condition. But in the majority of cases, a resume of the local user's activities will not overload the system, so the user may experience a degraded system

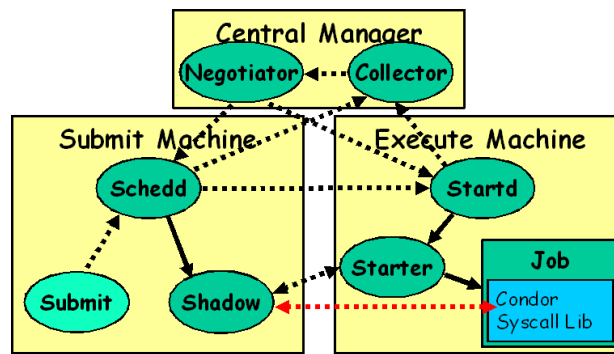


Figure 2.3: Architecture of Condor (from [Mil04])

performance for the time the SGE job still runs. A checkpoint/migrate action is only triggered on heavy local user activity, which causes the system to become overloaded. This quite static and inflexible mechanism may be acceptable in some environments, but falls short of one's expectations towards a general solution.

So although a traditional DRM system like SGE may be tweaked to perform opportunistic high-throughput computing, in practice there are some problems with this approach. These problems come essentially from the fact that systems like SGE have never been designed to be used that way.

2.4.2 Condor

The Condor system [Con05] has been specifically designed for opportunistic high-throughput computing. It has an overall architecture that is quite similar to generic DRM systems like Sun Grid Engine, but there are a lot of differences in detail. As outlined in Figure 2.3, Condor differentiates between four types of nodes and many services that run on them. The central manager node controls the whole pool and is the home of the *collector* service, which updates a central repository of static and dynamic system information, represented in the form of *classified advertisements (ClassAds)*. The *negotiator* service arbitrates resource offers (as presented by the collector) to resource requests by the *schedd* services located on the submit nodes. An execution node is where Condor jobs are actually run. There is a *startd* service running on each of them, which advertises node information to the collector and receives job start requests. A *starter* service sets up the sandbox environment for every job that is going to run on that node. At a submit node, where users log in, a *schedd* service tries to find resources for all jobs that have been submitted from this node. Furthermore there is a *shadow* service running for each started user application, which acts as an endpoint for intercepted system calls and executes them locally in the user's home environment. There is the

additional possibility of a checkpoint node, where a *checkpoint* service stores process images. In this case process suspension and migration are enabled.

The flow of jobs through Condor is in principle like the flow presented above for SGE. We will only describe some important differences in the following. *First*, Condor is slightly less centralised since it starts a kind of personal scheduler which monitors all jobs run by a particular user. The job is given to the scheduler in form of a configuration file, which is not as flexible as a shell script, but allows for easy extraction of file staging information. *Second*, the Condor negotiator does not take the availability of any resource for any type of job as granted. It rather allows for fine-grained access control in the form of ClassAds. These specify the type of jobs (run time, size, originating user/department, etc.) a given resource is willing to accept. *Third*, application programs commonly need to be linked with special libraries provided by Condor. This allows the attachment of an interposition agent at run time to perform automatic checkpointing, interact with the Condor master to write out a checkpoint just before an application process is killed, or forward certain system calls back to the user's shadow service. *Fourth*, Condor does not come from the assumption that application jobs run to completion. This is probably the most important difference to more traditional, static migration schemes: Both when local activity is detected and when the constantly updated priority scheme starts to favour another job, a job is checkpointed and then killed. It may later be resumed due to new free resources and/or priority increase.

2.5 Discussion

The systems mentioned above are very mature and provide a quite good solution in their respective domain. But one must consider that both systems apply a set of assumptions and restrictions: "Native execution" systems like SGE work best on dedicated resources, since opportunistic access to user workstations with their unpredictable idle periods cannot be handled well. Additionally, there must be some sort of centralised user identity management, which provides an account for every user on every computer. "Sandboxing" systems like Condor usually do not have these two restrictions. But other restrictions apply: To perform sandboxing and interposition, the Condor libraries need to be tightly aligned with the underlying operating system libraries, which causes Condor to be rather picky about the exact OS versions it is willing to run on. To plug the interposition agent into the application, application programs must be re-linked, which in practice usually means that one needs to be able to compile them, which implies source code access. Both systems require a separate version of the application for each possible target OS, although the final decision which one to run can be deferred after the scheduler has

2 Background and Related Work

decided about the execution resources. Once started, applications cannot be migrated across different platforms.

While both systems and many similar ones are successfully used in a variety of settings, there are still situations where some of the requirements are not met. In our experience, especially in the engineering sphere, the use of the discussed systems is not reasonable. Many laboratories nowadays have a Beowulf-style cluster running Linux that serves as the main platform for running larger simulations and calculations, which are often high-throughput applications. On the other hand, there may be a large number of workstations running Windows, which are idle during certain periods. Although both platforms feature the same hardware, it is not easy to write programs which run smoothly on both. For many typical engineering problems there are pre-packaged, commercial solvers available (e. g. FEM solvers), which usually must be bought separately for every platform they need to run on. In summary, we may find an environment which is not well covered by traditional systems. The HARPY project aims to fill this gap: to dynamically expand the capacity of a “core” Linux cluster with the resources of idling Windows workstations and to provide transparent migration between these platforms.

3 Architectural Overview of HARPY

3.1 General Design Considerations

The key idea of HARPY is that application processes, which are originally started on dedicated machines of a core cluster, are eventually migrated to user workstations which run idle. After applications have been migrated, they still see their original, dedicated cluster environment. So we do not care about “job start” or “file staging” on user workstations. Instead, the DRM system controls the core cluster on which all applications are started and on which all data access takes place. Normally, the DRM scheduler starts just enough jobs to saturate the capacity of the cluster with application processes. As user workstations find themselves running idle, they “extend” the core cluster by joining the pool and receive processes from the cluster via the migration facility. Thus, the pool dynamically gains computing capacity and allows the DRM system to start even more jobs. When local workstation users resume their activities, workstations leave the pool by pushing the processes back into the core cluster. Of course, sometimes this leads to a too high load on the core cluster. In those cases, some processes need to be temporarily suspended. But when either idle workstations join or when other processes complete, suspended processes will be reactivated again.

To provide a consistent environment even when an application process has been migrated to a user workstation, some sort of *Interposition Agent* is needed to maintain this illusion. Every interaction of an application with its environment has to be suitably translated. Experience with Condor [TL03] has shown that attempting to achieve this effect with internal interposition agents is complicated, fragile and sometimes quite slow, while kernel-level interposition agents generally do not suffer from these disadvantages. But to require each user to install a customised, interposition-enabled operating system kernel on his workstation is largely impractical. Instead we propose the use of *virtual machines*¹, which allow a customised operating system to co-exist with the unmodified original operating system side by side. While specialised virtual machines for distributed systems have been around for a long time [DPG98], recent trends on systems virtualisation tend to favour

¹ Virtual machines (VM) provide a machine abstraction on top of the “real” machine. This VM is usually able to boot a separate operating system instance.

standard components inside virtual machines for improved application compatibility. For HARP Y this means we need a customised standard operating system kernel which can act as interposition agent and is run inside a virtual machine. For HARP Y to be a minimal-invasive solution, the virtual machines need to run solely on top of an unmodified host operating system, unlike hypervisor systems (e.g. Xen [BDF⁺03]), which place themselves as lowest layer between the hardware and *all* running OS instances. A user-level Linux implementation meets exactly our requirements: it starts a complete Linux kernel as a user process under the host operating system. This Linux kernel includes all necessary modifications to support transparent process migration. Thus we get both: no modification of the host operating system, and still the full power of a customised kernel.

3.2 Components

Two of the three main components found in a HARP Y environment are quite natural: We need a set of execution nodes, named *Clients*, which run a virtual Linux machine on top of Windows and receive migrated processes; and a set of nodes running Linux, where the executions starts and where processes are sent from. Those are named *Master Execution Nodes*. Clients and master execution nodes form, together with a capable network in between, a HARP Y *Pool*.² The third main component is not so obvious: A central control instance, the *Configuration Server*, is needed to configure and control all pools that are located in a given network. This is necessary, because a client does not initially know to which pool it belongs and where the Master Execution Nodes reside. So it first needs to contact a central control instance, the named configuration server.

The most important services running on each of the specified components, as shown in Figure 3.1, are described as follows.

3.2.1 Client

The Client software consists of three main services. The *Linux VM* is central to the whole system and provides an execution container for application processes. The *Configuration Agent* is responsible for the automatic configuration and initialisation of the Linux VM as instructed by the configuration server. It updates the static pool information (processor speed, available memory, etc.) and the dynamic values for the load sensor as well. Finally, there is an *Idle Monitor* on each client, which activates the client if there has been no user activity for some period of time.

² Similarities to the pool concept of Condor [Con05] are intentional.

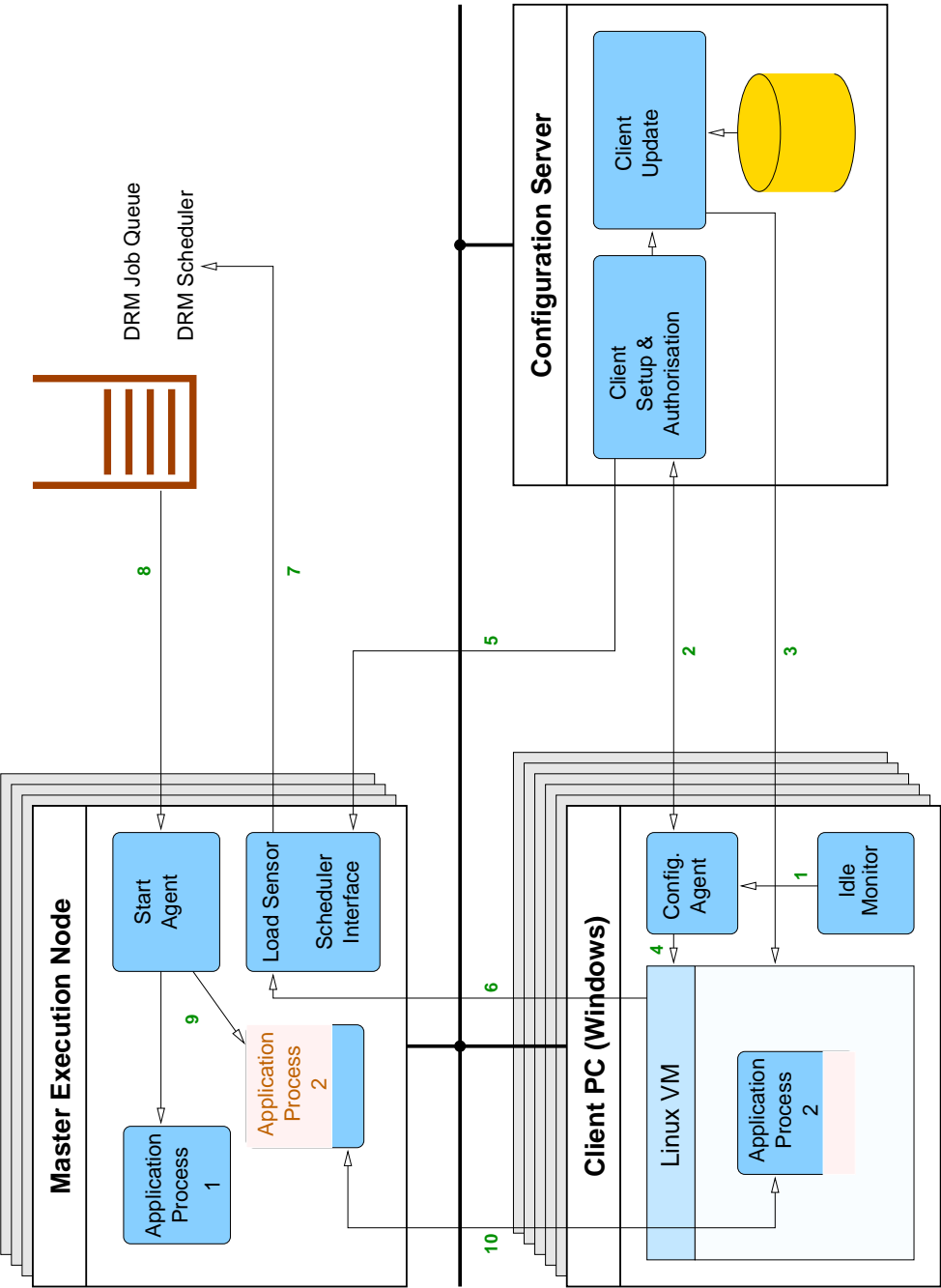


Figure 3.1: General architecture. For the small digits refer to section 3.3.

Although the Linux VM can be a full-blown Linux system in principle, it is kept rather minimalistic. Jobs are exclusively started on the master execution node and are migrated afterwards,³ so all necessary utilities, libraries, and job control/monitoring facilities are not required on the Linux VM. Quite the contrary, it needs to be hardly more than just a “naked” Linux kernel, enabled with migration mechanisms to receive application processes and execute them on behalf of the master execution nodes. Since any environment interaction of the application processes is forwarded to the master execution nodes, the only utilities required to be locally installed are those needed for the management of the Linux VM itself, i. e. to query load values and to receive messages from the configuration agent. This should lead to a very small installation: we expect the Linux VM installation image to be less than 10 MB.

The configuration agent is mainly a client to the configuration server. It first locates the configuration server and then performs a small handshake in order to authenticate and register. It must be able to receive some configuration values from the configuration server like network setup information etc., to adjust the Linux VM’s configuration files accordingly. The configuration agent then boots the Linux VM and waits until it is signalled by the idle monitor to shut it down again.

The idle monitor is even more straightforward. Since memory considerations (see Section 5.2) give the advice to start the Linux VM only when there is absolutely no user activity on the client machine, there is already a Windows service which can be hooked into: the screen saver. So the idle monitor is just an interface between a screen saver application and the configuration agent. Of course, more elaborate idle monitors are also imaginable.

The whole client setup is discussed in Section 4 in more detail.

3.2.2 Master Execution Node

Master execution nodes are part of the core cluster. They are dedicated machines which provide a home environment for user applications. Although we think of a core *cluster* here, there are also scenarios imaginable which consist only of a single master execution node (for example, a compute server). This, however, would make no difference to the concepts presented here.

Master execution nodes are integrated into a conventional DRM system and feature two main services. The DRM runs a *Start Agent* which forks off application processes from the DRM system’s job queue. Many DRM systems provide a programmable interface by which they pick up current load metrics from each execution node. This interface is operated by the *Load Sensor*. Such

³ In this respect, HARP Y is quite similar to the *BProc* distributed process space [Hen02]. Since all job starts occur on the master execution nodes and processes “take their environment with them” as they migrate, there is no need to maintain a full system installation on the client nodes.

programs normally query only the execution node's operating system, but for HARPY this needs to be modified in order to provide load metrics according to the load of the whole pool.

When the start agent executes a user application, it runs initially on one of the master execution nodes, thereby increasing its load. This will be noticed by the migration facility, which constantly monitors the whole pool and tries to balance the load over all nodes in the pool. Since there is now a load imbalance between the master execution node and the clients, application processes will be migrated to the least loaded node. Only a thin system context from the migrated process remains on the master execution node, serving as an endpoint for forwarded system calls. Notice that the process migration does not happen immediately, since the migration facility needs some time to detect the load imbalance. But often applications begin with a phase of heavy input/output activity. Data access is performed more efficiently on the master execution node than on the client, because system calls need not be forwarded. Thus, it may be advantageous to defer the application's migration up to the point when it enters a processor-bound computation phase. Some migration facilities, for example openMosix, can detect and handle this automatically.

The load sensor needs to compute load metrics for the whole pool, in order to guide the DRM scheduler in its decision when to start jobs. Two important factors are processor and memory load on all clients. While it is quite easy to decide about job start from a processor load perspective (start additional jobs until all nodes have a sufficient CPU utilisation, e. g. more than 90%), memory utilisation is not so easy to determine. Consider an example where we have three nodes with 400 MB of free memory each. Although calculatively within the total memory limit, it is not possible to start four applications requiring 300 MB of memory each without driving at least one of them into thrashing.⁴ Instead, the number of concurrent jobs needs to be limited so that each job fits into *one* machine's memory. Another important factor is the current network bandwidth utilisation at the master execution node. It would not be wise to let the DRM start a new job that is likely to migrate soon, when there are already too many migrations going on. In this case there would be no network bandwidth left for another migration, and forwarded system calls from the clients would be delayed. The processor, memory, and network utilisation of all nodes, which are already monitored by the migration facility, and the processor, memory, and network utilisation of the master execution nodes, which are monitored by the load sensor, can be combined to form an overall pool utilisation measure. This value, together with a threshold value

4 From [Ray96]: "thrash *vi.*: To move wildly or violently, without accomplishing anything useful. Paging or swapping systems that are overloaded waste most of their time moving data into and out of core (rather than performing useful computation) and are therefore said to thrash."

computed from the maximum capabilities of the pool, is sent to the DRM scheduler.

Sometimes pools increase their load without starting new jobs, for example when nodes leave faster than applications finish. Eventually, overload conditions occur on the remaining nodes. This needs to be detected, either by the load sensor on the master execution node which queries the migration facility, or on the clients which notify the load sensor themselves. In any case, the DRM scheduler needs to suspend one or more jobs. Suspended jobs receive no processor time until they are continued again, so they can completely go into the master execution node's swap space. Alternatively, when there is a checkpointing service in effect, a checkpoint of those jobs can be taken and saved to disk.

3.2.3 Configuration Server

On the configuration server there is a client setup & authorisation service and a client upgrade service. The *Client Setup & Authorisation* service is a central instance contacted by every client on startup. The client is authorised with a challenge/response protocol to ensure that the software installation on the client has not been tampered with. Then there are several configuration settings to be transmitted to the client depending on the class a client belongs to. Namely, the configuration server decides on the pool (if there are several), tells where the master execution nodes for that pool reside and which network setup to use. Client classes need to be configurable depending on IP address ranges or host names/domains. Note that after client network setup, which is explained in Section 4.1.3 in detail, each client's official network address is complemented with a private network address. The configuration server needs to keep track of these associations, since it has to use the right address when communicating with different services. For example, when a client is finally integrated into its pool, the master execution nodes need to be informed about the new client's private network address.

In case that the client needs a software update, the *Client Update* service is called by the setup service. Since we use kernel-based process migration facilities, it is critical that the Linux kernel versions at the clients match those at the master execution nodes. In reality we might encounter several pools whose core clusters might use different kernel versions. To cope with this, the client update service needs to have a repository of several installation images of the Linux VM. After the client setup service has classified a new client, it queries the client's software version and compares it to the software version required for the pool where the client is going to be integrated. In case of a mismatch, the client update service is instructed to transfer a new installation image to the client, where it is installed automatically.

3.3 Events in HARPY

To gain a better understanding of the interrelations between the various components and their services in HARPY, we provide an alternative view on the architecture by studying the flow of actions on three key events: the addition of a node to the pool, the start of a new job and the eviction of a node. The numbers in the text reference to the arrows in Figure 3.1 on page 23.

Node Addition The term *Node Addition* refers to a node being added to a pool. The machine may be already running for a while, but once it becomes idle the screen saver gets activated by the host operating system. It then starts the configuration agent (1). The configuration agent locates the configuration server, either via broadcast or by querying a pre-configured address, and authenticates itself (2). The client is classified by the client setup service. If the client's software is found to be outdated, the configuration server sends an updated image to the client (3), which is installed by the configuration agent. After that, it retrieves its configuration settings. Now the client is ready to run. The Linux VM is started (4) and performs its network setup (cf. Section 4.1.3). In the meanwhile, the configuration server updates the static node information at the master execution node (5), which is in turn passed to the DRM scheduler by the scheduler interface. After successful initialisation, the Linux VM contacts a master execution node itself (6) and completes its node information by providing dynamic values, e. g. system load. Now the client is ready to accept application processes.

Job Execution With the information provided by the configuration server (static node information), the Linux VMs (dynamic load information) and the master execution nodes (static and dynamic information), the DRM scheduler knows enough to estimate the free capacities of the pool (7). As part of its normal operation, it starts new applications from the DRM system's job queue if applicable (8). On a job start, the DRM scheduler instructs the start agent on a master execution node to fork off the application process(es) (9). Most likely, the load on the master execution node is now higher than the load on other nodes in the pool. So the process migration facility starts to balance loads. The application process may eventually be transferred to a client node, or it may stay on the master execution node, depending on the situation. In any case the process migration facility ensures that I/O is done the same way, maintaining the illusion of running on the master execution node all the time for every process (10).

Node Eviction When the local user resumes activity, the node has to be evicted. The screen saver terminates and sends a shutdown signal to the con-

figuration agent. The configuration agent signals node eviction to the configuration server (2), which forwards it to the DRM scheduler by the master execution node's scheduler interface (6, 7). Concurrently, the configuration agent forwards the shutdown signal to the Linux VM. This terminates the process migration facility on the local node, causing its remote processes to be migrated back into the pool. After it has been evicted, the Linux VM terminates itself. Since the presence of the Linux VM degrades the workstation's performance as perceived by the local user, the node eviction process should complete as quickly as possible. While it can take a substantial amount of time to migrate processes off a Fast Ethernet connection,⁵ with Gigabit Ethernet this presents no particular problem. The shutdown of the Linux VM itself is expected to take less than 5 s, since there are hardly any additional processes running inside.

What happens to the now "orphaned" processes? The process migration facility tries to balance load again, migrating them to other client nodes or master execution nodes. Especially when several nodes have to be evicted in a short period of time, this may lead to a severe overload of the remaining pool. This condition has to be detected by the load sensor, alerting the DRM scheduler. The DRM scheduler then decides upon one or more application processes being suspended, so that they are "parked" on the swap partition of their respective master execution node. Thus, overload situations are handled quickly enough that no node remains in an unproductive overload state for more than a process migration time. Suspended processes are resumed as the pool load decreases in consequence of other terminated application processes or added client nodes.

3.4 Summary

The architecture of HARP Y provides mechanisms for the detection of under-utilised workstations, the automatic configuration and update of client software, the isolated and transparent execution of application processes, and the interaction with a distributed resource management software. With these components in place, we are able to conveniently take advantage of otherwise idling computing power without requiring the user to specifically design his applications towards this property. The technically most demanding part of the whole architecture is the client. It is going to be explained more thoroughly in the next chapter.

⁵ For example, the migration of a 200 MB process image, which is not unusual for a scientific application, takes in the best case about 20 s.

4 Client Architecture

The client software consists of several modules, which will be explained in this chapter. We put a particular focus on the interaction between these modules. Also, both the Linux VM and the configuration mechanism are going to be explained more in detail.

4.1 Construction of the Linux VM

4.1.1 Running a Linux Virtual Machine

In order to embed Linux processes into our Windows-based client machine, we need a virtual machine, where we can install a Linux kernel specifically tuned for HARPY operation. It is not feasible to reboot the whole machine for cluster operation as proposed by systems like ClusterKnoppix [Van04], since all running local programs would be closed then. Instead, we need some sort of virtualisation that does not affect the underlying system in an undue fashion.

To realise a virtual Linux machine on top of an unmodified Windows host operating system, there are three general solutions:

1. Emulate a complete PC hardware in software and boot a guest operating system on top of it.

This approach is used by commercial PC virtualisation products like VMware™ [VMw04] or Microsoft's Virtual PC™ [Mic04], and several free projects (like QEMU [Bel04] etc.). Although these differ in several important details, they all provide “virtual hardware”: A complete PC with a set of standard components (video adapter, network adapter, etc.) is emulated, which allows to boot a completely unmodified guest OS on top. These systems come with a noticeable performance loss, since the emulation of hardware in software always causes significant overheads.¹ Additionally, the emulation costs increase when the instruction set of the virtual machine does not match its physical counterpart, since then the CPU must be emulated as well.

¹ Unfortunately both named commercial vendors prohibit the publication of benchmark results in their End User License Agreements, so that there is only little independent performance data available. An older VMware version without this restriction is benchmarked in [BDF⁺03].

4 Client Architecture

2. Run a specially prepared *User Mode* guest operating system.

Here the guest OS (Linux in our case) needs to be adapted so that it does not attempt to drive the underlying hardware platform directly, but rather uses the services of a host OS as its platform. This approach is limited in this respect that it cannot virtualise the instruction set architecture, hence it is restricted to the use of different operating systems on the same hardware architecture. The big advantage, however, is that it has only little CPU overhead.

3. Emulate the guest system at the API level.

To get the best possible performance, there are also emulators which do not use a guest operating system kernel altogether. They link applications with special interposition libraries, which transform API calls from one operating system to another, then forward them to the original OS, for example like CygwinTM. This approach gives no opportunity for guest OS modifications, since we do not even run a guest OS kernel.

While the first approach involves unacceptable performance degradation, and the third withholds us from the possibility of guest OS kernel modifications, the second one presents a good compromise between speed and flexibility. Several systems have been devised to provide a user-mode port for Linux running on top of a variety of host operating systems. The *Cooperative Linux* project (or short *coLinux*) [Alo04] provides a set of Linux kernel patches and associated helper programs to run a Linux kernel as a normal task within a host operating system.

Cooperative Linux running under Windows² works through the cooperation of several key parts. First, there is a Linux kernel which is modified in such a way that it accesses the underlying hardware only with special precautions. Second, the Linux kernel is wrapped into a special device driver, called `LINUX.SYS`, which embeds the whole running Linux system, consisting of the kernel together with all its user space processes, into the Windows kernel space. Since the memory accessible from the Linux kernel is cleanly separated from Windows, the `LINUX.SYS` driver provides a single point of communication through a passage page. This ensures system security: Processes inside the VM cannot access anything from the host OS. Third, there is a monitor daemon called `colinux-daemon` running in Windows user space. It interfaces with both the driver and the user, providing a console. Network communication is currently handled via an auxiliary networking daemon which is attached to the monitor daemon (see Section 5.3 for details). Note that the term “user mode Linux” is strictly speaking not an accurate description for

² Cooperative Linux also provides a Linux-Linux setup which embeds a guest Linux system into a host Linux system. This setup is not going to be considered here.

coLinux, since coLinux does not run entirely as a user-mode-only process. Despite that it therefore needs administrator privileges to run, it behaves in every other respect like a user mode Linux implementation. In particular, it requires no modifications to the host operating system apart from the installation of a special driver, and provides high emulation efficiency.

4.1.2 Integration of a Process Migration Facility

There are several systems subsumed under the term “single system image” which provide a facility to migrate running processes from one Linux system to another. This particular research topic has gained attraction recently, so the list of some long existing systems like Mosix/openMosix [BGW93, BL98, B⁺04] and OpenSSI [W⁺04] has been supplemented with newer developments like Kerrighed [VLR⁺03] or Plurix [GSFS04].

Several of these SSIs are eligible to be combined with coLinux, despite these systems differ in a lot of ways from each other. To decide which one is best suited, we establish some requirements to the process migration facility. First, it must be possible to add or remove nodes on the fly, without restarting the whole cluster. Second, the source code must be freely available, in order to integrate it into coLinux and publish the results. Third, the required modifications to the existing Linux kernel code base should be as lightweight as possible in order to reduce the potential for conflict. The migration functionality should be performed within separate kernel modules as much as possible. Fourth, we need some form of caching mechanism for heavily used system calls to avoid the network latency on each and every system call when it gets forwarded to the original execution host.

Another thing to consider is the main entry point for kernel modifications. CoLinux relies heavily upon the low-level memory handling routines to separate access permissions inside the Linux VM from those of the host OS. Hence, a process migration facility that also relies mainly on memory handling (like Kerrighed and Plurix do) is harder to integrate than one which mainly attaches to other kernel mechanisms. For example, Mosix and openMosix both base on manipulations on the process table, which is mainly independent from the low-level memory management.

With these requirements applied, we currently end up without a process migration facility that is meeting them fully. But several come close: openMosix provides nearly all but has no caching. Until now, Kerrighed supports no dynamic node addition/removal, but seems to be quite promising in the other areas. OpenSSI and Plurix have been found to be too restricted for our purpose: OpenSSI demands heavy adaptations on the core cluster as well, which may not be feasible if the core cluster does not already run OpenSSI, and Plurix is quite Java-centred. We are currently preparing an openMosix-

based prototype [SS05], but keep an eye on Kerrighed, since it has an advanced caching mechanism and is thus promising better performance.

4.1.3 Network Setup

Since HARPY is intended to operate on the client machines as unobtrusive as possible, all input/output is performed via the network interface and the local hard disk is used only as swap space. Thus good network performance is critical. We need both low latency and high throughput: On one hand, latency is important since system calls must be forwarded to the master execution node where a application process has originally been started.³ Since system calls are blocking, the application process is suspended until the response to the system call arrives back. On the other hand, when we migrate large process spaces, we want to utilise the full bandwidth of our networking hardware to make the migration as quick as possible. This is particularly important on typical 100 Mb/s Ethernet office networks.

Several factors contribute to network performance. There needs to be a high-performance implementation of the network protocols, which is quite good in current Linux kernels, and a quick handling of network data at the interface between host and guest OS (which current state is examined in Section 5.3). Since the Linux VMs do not have direct access to the network hardware, there needs to be some sort of virtual or logical network setup to connect the Linux VMs on the clients to each other and to the master execution nodes. The setup of this logical network must not degrade network performance, too. We must also consider that the Linux kernel running inside the VM has its own full-blown network stack that is not prepared to interact with a host operating system's network stack on the outside. To cope with this, there are several possible schemes:

Network Address Translation (NAT) In the simplest case, the Linux VM holds a private IP address and the host operating system is configured to masquerade outgoing connection requests from the Linux VM via NAT to feature the "official" IP address. Thus, requests appear to come from the host OS itself. Responses to an already established connection are translated back to the Linux VM's private IP address. This scheme involves some (albeit no big) processing overhead on the host OS's side. A serious disadvantage is that all connections must be established from within the Linux VM and there is no way to contact the Linux VM spontaneously from the outside. Since this is necessary for the operation of the process migration facility, a NAT scheme is no option.

³ In fact, with openMosix nearly *every* system call is forwarded.

Network Address Translation With Port Forwarding In this refined version of NAT the host OS is instructed to forward incoming connections on some pre-defined ports to the Linux VM. Thereby the Linux VM is able to react to spontaneous requests from the outside. While this is an option in principle, it may conflict with certain process migration facilities. For example, openMosix expects the same IP address both on its local interface configuration and as entry in the node table distributed to all nodes. Here the use of NAT plays a trick on us: the official IP address must be used to contact the Linux VM from the outside and hence inserted into the configuration file, but the Linux VM finds arriving network packets to have its private IP address as recipient. This effectively stops openMosix from working. Additionally, the automatic setup of the firewall rules required to accomplish this is somewhat difficult to achieve on most Windows versions.

Shadow Network When there are no routers within a pool (i. e. all nodes are on the same subnet), we can just forward unmodified packets from the Linux VM directly to the physical network interface and thus expose the private IP address of the Linux VM to the outside network. When all clients are doing the same, the addressing scheme is consistent again, but of course packets cannot pass any router unless the router is specifically configured. This may or may not be a problem: for pools spanning several subnets it is clearly unacceptable, while on the other hand it is quite easy to setup when all clients and master execution nodes of a pool share the same subnet. Additionally, the pool is completely isolated from the rest of the network. This scheme is practical for automated setup and has nearly no processing overhead, so we expect the best performance here.

Virtual Private Network To add a subnet-crossing capability to the private IP addresses used by the Linux VMs, packets originating from the VMs can be wrapped into additional IP headers containing the official IP addresses of the client PCs. This way a virtual private network is formed. This involves additional processing overhead and thus latency, but gives us the highest flexibility. There are two sub-variants: first just plain IP-in-IP tunneling, second the use of encryption to protect the whole traffic [Y⁺04, ST⁺04]. This, of course, adds an even higher processing overhead.

In summary, the shadow network scheme has the lowest overhead and is consequently the best choice as long as all the nodes of a pool are on the same subnet. In the case of several subnets, one has to ponder between a pool splitting, so that there is one pool for each subnet and we can use the shadow network again, and the VPN scheme. The VPN involves a certain overhead, but is quite flexible and thus the second best choice. NAT with port forward-

ing is too complicated to set up, while NAT without port forwarding is out of question.

4.2 Installation, Configuration, and Update

4.2.1 Installation Procedure

Besides the Linux VM there are two additional services on each client: the configuration agent and the idle monitor. One important task of the configuration agent is to keep the software installation up-to-date. In order to understand the requirements of the installation and update process, we first take a look at the items that have to be installed on the client. The installable items include:

Client Helper Programs Some binaries built specifically for HARPY: the idle monitor (screen saver) and configuration client. These items need not to be updated often.

coLinux Daemons and Utilities The Linux VM requires a set of associated programs, which include the `LINUX.SYS` driver, a system monitor and others. These items only need to be updated on every new release of coLinux.

Networking Utilities In order to connect the Linux VM to a virtual private network, some auxiliary programs are required too, for example a TAP driver.⁴ Although these utilities have rather long release cycles, they must probably be updated on changes with the network setup. We also note that some of these need an elaborate setup procedure, so it would be better not to change them often.

Linux Kernel and Filesystem To start the Linux VM, images of the kernel and at least the root filesystem are needed. The exact contents are highly dependent on the configuration of the pool's master execution nodes. So we expect that these files need to be updated often.

Linux Swap File This can be generated automatically, perhaps even dynamically in the moment the Linux VM starts. So we do not need to fetch it from an installation server.

All of the items listed above need to be present on a working client installation. In order to prevent functionality duplication, it is advantageous to keep the initial installation image as lightweight as possible and fetch most of the

⁴ A TUN/TAP device provides a virtual Ethernet segment which connects all programs using a VPN on a host with each other and with the VPN driver.

items during the update phase of the client configuration. So the only item which is required to be in an initial installation image is the small collection of client helper programs. During initial installation, a working directory has to be chosen, probably by the user. The client helper programs have to be installed there. Then the idle monitor has to register itself with the host OS. After that, the local user should probably not be bothered with HARPY any more.

The rest of the installation as well as any subsequent client update works through the same mechanism: the configuration agent asks the configuration server about all required items together with their version, represented probably as a file list in combination with MD5 hashes of the files' contents. It is important for the configuration agent to check for exact version match and not a greater-equal relation found in most common packaging systems. Otherwise it would be difficult for a client to switch between different pools where each pool requires a certain version of the Linux VM. After having determined which items need to be updated, the configuration agent requests those from the client update service, possibly via a common protocol like FTP or HTTP. Each item is accompanied with its own setup script which is meant to perform all necessary setup operations unattended.

4.2.2 Configuration Issues

Besides doing installation and update, the configuration agent gathers pieces of configuration information from various sources and feeds them to a number of services. In the following, we will explain some key pieces of this configuration data flow.

Configuration Data Sources The configuration agent receives its data from two main sources: configuration information that can be determined locally and those to be received from the configuration server. Local configuration includes static machine information like processor architecture, the amount of local memory, and the amount of free disk space. An important piece of information is the location of the configuration server. Its host name can either be queried from the user during initial installation, or can be pre-set within the installation image. Alternatively, it can be dynamically found using a broadcast on all local interfaces. Depending on the network environment, each method has its advantages and disadvantages: A statically set host name is not flexible enough in the case that the configuration server should change some day (but there is the possibility to set up a DNS alias), while broadcasts are generally reaching only the current subnet.

Configuration data obtained from the configuration server includes the pool a client is assigned to, the addresses of the master execution nodes within

that pool, and the network setup to use. Additionally, the configuration agent retrieves a list of software and associated version information, which serves as a basis for the client update.

Configuration Data Targets Some pieces of configuration information are processed locally by the configuration agent, including everything related to the client update. Others affect the Linux VM. Based on the memory size, a swap file has to be created. The coLinux daemon must be informed about the file names of the kernel image, the root filesystem image and the swap file. In case of a VPN network setup, there are some VPN services which need to be configured and started. Furthermore, information regarding the master execution nodes and the network setup have to be communicated with the Linux VM. This is not so easy, since usually the Linux VM is isolated from the host OS. Possible solutions include the use of an auxiliary network device which provides some sort of “enhanced loopback” to connect the Linux VM to its host OS using a network protocol, or the use of a special configuration file which is laid out in such a way that it can be written by the configuration agent and is read during startup of the Linux VM. Since we conjecture that no two-way dialogue between the Linux VM and the configuration agent is necessary, we currently favour the configuration file since it is simpler.

To sum up, the configuration agent is the central switch on the client which acts as a broker between the Linux VM, the client’s host OS and the various services located at the configuration server. Of course it is impossible to consider every interaction that it has to do, so the implementation we are currently working on emphasises flexibility and extensibility. But once the Linux VM is set up, the configuration agent has no more duties (except to wait for the shutdown signal), passing the focus to the inner workings of the Linux VM. In the next section we will analyse factors towards a good performance of the Linux VM.

5 Performance Evaluation

Although it is too early to attempt a full performance prediction based on a model of the whole HARPY system here,¹ we will examine the performance of various key parts of the client architecture. This is necessary in order to spot possible performance bottlenecks. Note that we will only consider the performance of application programs executing on the client. The consideration of possible scalability issues with the configuration server or with the master execution nodes will be deferred to a later project stage as well.

To get a prediction about possible performance implications, we compare two scenarios: in the first one the application is running natively on either a standard Windows or Linux system. Compared to that, in the second scenario the application is running on the same hardware, but with HARPY, i.e. within a Linux VM on top of a Windows system. This comparison is roughly the one between the “native execution” approach from Section 2.3 and ours. Thus we try to measure how efficient the client hardware’s utilisation is.

5.1 Processor Usage

It is clear that some features of HARPY, like forwarded system calls, will have performance consequences. But with these stripped of, is there still a performance hit just by using coLinux? To answer this question, we have conducted a small test with the POV-Ray benchmark [EMTT03, POV04]. This benchmark features a somewhat longer (about 1.5 h) calculation of a ray-tracing image involving nearly no input/output and only a few system calls.

Figures 5.1 and 5.2 on the following page show the results for two representative benchmark runs on the same hardware (a Pentium-M 1.4 GHz laptop), the first one with a native Linux kernel booted and the second one within the Linux VM (on top of Windows), but without any process migration enabled. The two runs perform almost equal, with only a small performance hit during the coLinux run. Additional test runs, also on different hardware, have yielded quite comparable results. The efficiency is about 97.9%, which is fairly good. This seems reasonable, since coLinux has no kind of hardware emulation, there are just some additional background services running on the Windows platform compared to native Linux.

¹ A full performance evaluation will be provided at a later implementation stage when there is more experimental data available.

5 Performance Evaluation

Total Scene Processing Times				
Parse Time:	0 hours	0 minutes	2 seconds	(2 seconds)
Photon Time:	0 hours	1 minutes	22 seconds	(82 seconds)
Render Time:	1 hours	5 minutes	50 seconds	(3950 seconds)
Total Time:	1 hours	7 minutes	14 seconds	(4034 seconds)

Figure 5.1: Results of the POV-Ray benchmark on native Linux

Total Scene Processing Times				
Parse Time:	0 hours	0 minutes	2 seconds	(2 seconds)
Photon Time:	0 hours	1 minutes	23 seconds	(83 seconds)
Render Time:	1 hours	7 minutes	15 seconds	(4035 seconds)
Total Time:	1 hours	8 minutes	42 seconds	(4122 seconds)

Figure 5.2: Results of the POV-Ray benchmark on coLinux

5.2 Memory Access

While the first benchmark shows good results, one must bear in mind that this is a near-to-pure CPU benchmark: real applications quite often depend on the amount of available memory. Unfortunately, coLinux cannot use main memory the same way than native Windows applications can do. Both the Windows and the Linux kernel make use of the Memory Management Unit (MMU) to translate virtual addresses into physical ones. Since coLinux acts as a process within Windows' virtual memory, addresses would be translated twice: the first time by the memory management internal to Linux, the second time by Windows. This double page table lookup cannot be performed in hardware and is thus prohibitively expensive. In order to avoid that, coLinux currently allocates a large, contiguous memory block which is pinned² and deprived of the Windows memory management. Virtual addresses inside coLinux are directly translated into physical ones by the Linux kernel's memory management alone.

Hence on the start of coLinux, the user workstation's physical memory is effectively partitioned: A certain amount is set apart and acts as the only memory coLinux can access, while the memory usable for Windows is reduced to the remaining part. Of course, applications running inside of coLinux should be able to use as much of the physical memory as possible, since paging often involves a serious performance hit. This particularly holds true when the machine has been idle before coLinux starts. Nevertheless it is clear that we cannot allocate all of the physical memory to coLinux. To relieve this issue, some coLinux developers pointed out on the coLinux developers' mailing

² Pinned memory is taken out of control of the normal memory management, thus it is made unmovable and unpageable.

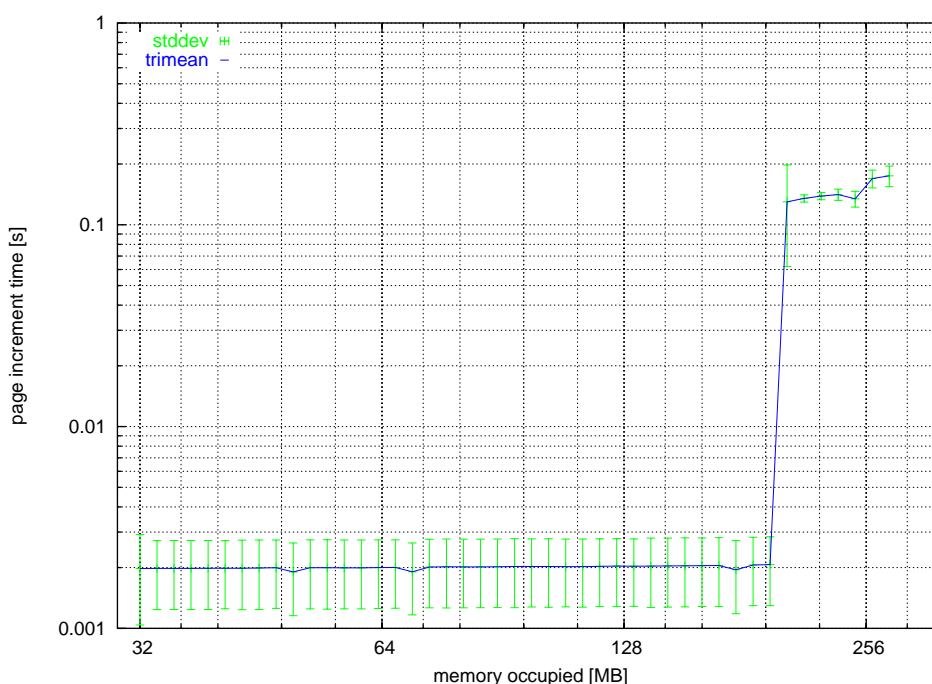


Figure 5.3: Memory access times on Linux

list [A⁺04] that paging from inside coLinux would not be equally expensive as paging under native Windows, since accesses to the coLinux swap file are cached by the Windows kernel’s buffer cache. This would lead to a much lower page miss latency and thus to a less severe performance burden.

To verify this claim, we have written a micro-benchmark which allocates a certain amount of memory, casts it into an array of `long` values and sweeps over the whole array incrementing each value. This way, it is ensured that every memory location is both read and written. While doing this, we measure the time this takes and divide it through the number of memory pages. Next, the allocation is enlarged via `realloc()` and the sweep begins again, up to a configured limit which is slightly larger than the physical memory size.

Results from three runs of this micro-benchmark are shown in Figures 5.3–5.5. Note the logarithmic scales on both the memory size and the page increment time. All runs have been performed on the same hardware, a laptop computer with a 1.4 GHz Pentium-M processor. The first run (shown in Figure 5.3) has been performed on a native Linux system. The solid line in the diagram denotes the trimean³ of eight consecutive runs, while the error bars

³ A trimean is a weighted mean where the middle 50-percentile of the sample counts double. This statistic has been chosen because the data sets are highly skewed.

5 Performance Evaluation

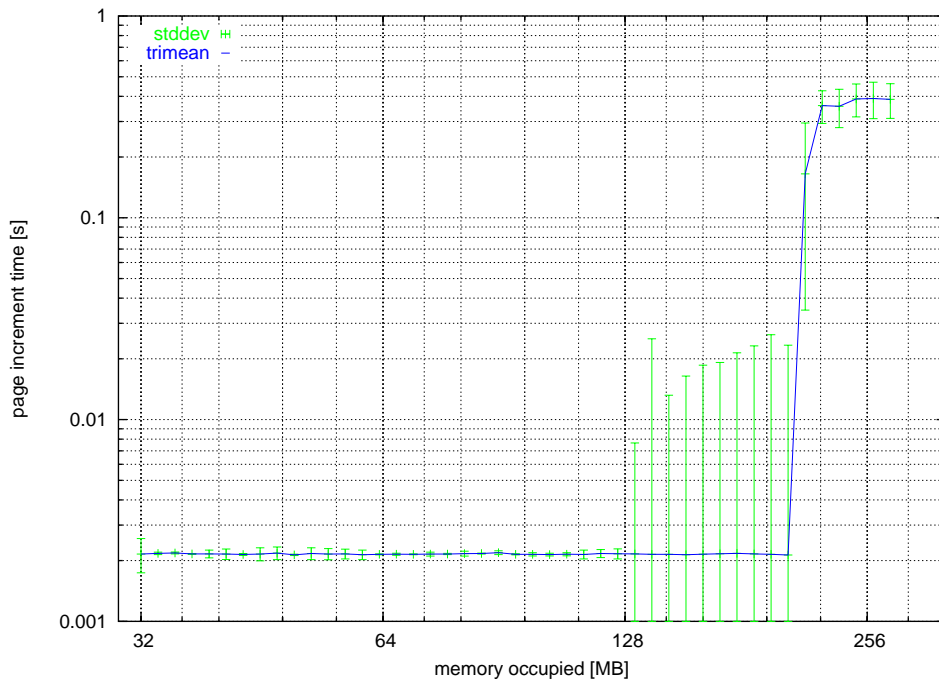


Figure 5.4: Memory access times on Windows

indicate the standard deviation. The diagram shows that the time for the increment of one page's amount of long values remains constant about 2 ms for a wide range of allocation sizes. The access times increase sharply at some point below the physical memory size of 256 MB, where the benchmark starts to push away other running operating system services and thus causes thrashing.

The second run in Figure 5.4, measured on a native Windows system running on the same hardware, shows no significant differences from the previous one except that even before thrashing occurs, access times tend to vary much more. There are also somewhat larger access times when the system is in thrashing state, indicating that the Windows kernel handles this state slightly less efficient.

The third run, shown in Figure 5.5 on the facing page, presents a completely different picture. In this case a memory partition of 64 MB has been created for coLinux and the micro-benchmark has been started within. We see quite comparable access times for allocation sizes up to 64 MB (excluding a small amount for the operating system services inside the Linux VM). When the allocation size exceeds this limit, the Linux kernel starts to swap. For allocation sizes between approximate 64 and 128 MB, access to the coLinux swap file, which is just an ordinary file on a Windows drive, is cached by the Windows kernel's buffer cache. Access times are indeed lower than those for

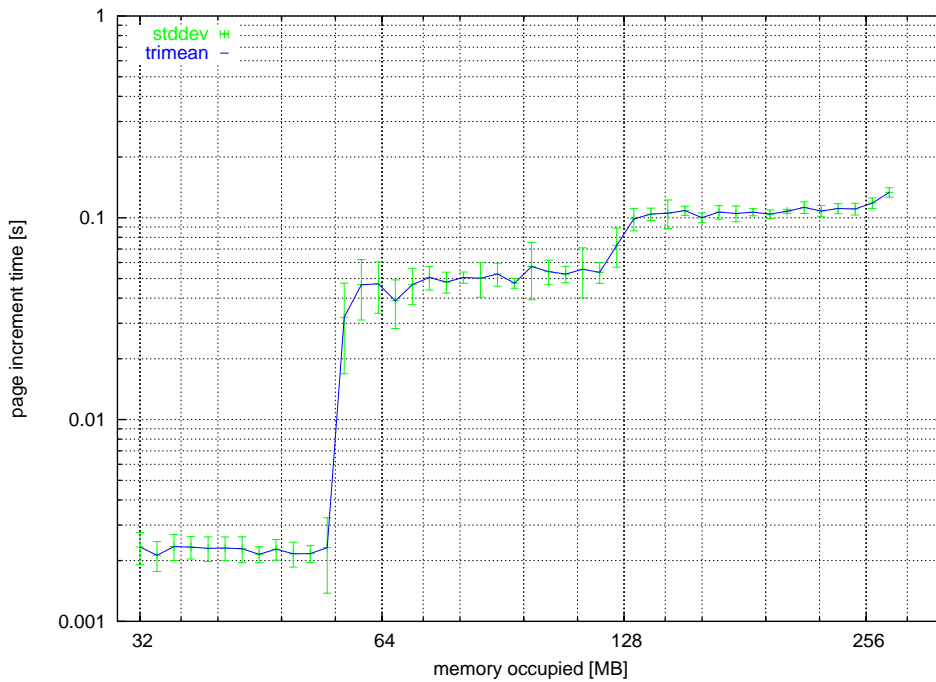


Figure 5.5: Memory access times on coLinux

direct disk access, albeit more than ten times higher than for small allocation sizes. When the allocation size is enlarged further, more and more paging requests need to be actually written to the disk. Access times finally approach the same range already observed on native Linux and Windows systems, and are mainly dominated by disk performance. Qualitatively similar results have been obtained during additional measurements on other hardware.

This indicates that although the use of the Windows buffer cache can attenuate swapping costs, these are by no means comparable to the case when memory accesses fall entirely into coLinux' own memory partition. According to the current state of coLinux, we can allocate roughly half of the physical memory. The use of the Windows kernel's buffer cache to speed up Linux' swap file access brings a performance improvement, but it is still too small. Further development effort has to be put into this issue, allowing more memory to be quickly accessible from within the Linux VM.

5.3 Networking Overhead

Besides good processor and memory performance, the network performance is crucial for the overall throughput of HARPY. We need both a low latency to cater quickly for forwarded system calls and a high throughput for timely

5 Performance Evaluation

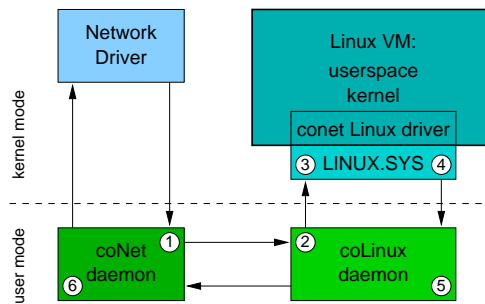


Figure 5.6: Networking data flow through coLinux. Numbers in circles denote the rough positions of the checkpoints in Figure 5.7.

process migrations which often involve the transfer of huge process address spaces.

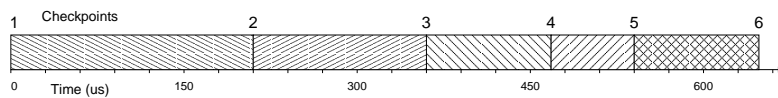
We cannot alter the network connection at the client (we assume Fast or Gigabit Ethernet), but we must be aware of possible performance bottlenecks in the forwarding mechanism which connects the Linux VM to the network. A data flow diagram is presented in Figure 5.6, showing how this is done in coLinux. Outgoing packets are sent to a “virtual” network device inside the Linux VM from where they are forwarded to the attached `LINUX.SYS` Windows driver (4) via a passage page. From there, they are sent to the coLinux daemon (5) which resides in user space. This control daemon dispatches all messages going into or out of the Linux VM. The coLinux daemon in turn pipes the network packets to a specialised coNet daemon (6), which handles all networking issues and emits the packets finally into the original Windows network device. Incoming packets for the Linux VM go the other way round (1–3). As result, the data path is rather long. This is mainly due to coLinux’ design decision to emphasise flexibility. In current versions of coLinux, typical latencies are quite high and the wire speed of a 100 Mb/s Ethernet cannot be reached, whereas both native Linux and Windows installations have much lower latencies and can reach link saturation.

To understand this issue, the coLinux code has been instrumented with six checkpoints that log time stamps as network packets are passing by. The rough positions of these checkpoints are indicated as circled numbers in Figure 5.6. Then the ping program has been used to send an ICMP PING request to the Linux VM and receive an ICMP PING reply back. After the time stamps have been transferred out of kernel space, we have calculated the time difference between each pair of checkpoints. To get acceptable average values, the experiment has been repeated about 50 times.

Table 5.1 and Figure 5.7 on the facing page illustrate these times on an 1.4 GHz Pentium-M laptop otherwise running idle. The time the Linux VM needs to receive and analyse the ICMP PING request and to assemble and send

Table 5.1: Processing times for network packets passing through the current coLinux networking architecture (cf. Figure 5.6)

From Checkpoint		To Checkpoint		Av. Time
1	co_win32_pcap_read_received	2	co_daemon_pipe_cb_packet	211 μ s
2	co_daemon_pipe_cb_packet	3	co_monitor_run	151 μ s
3	co_monitor_run	4	co_message_write_queue	109 μ s
4	co_message_write_queue	5	co_daemon_run(recv)	72 μ s
5	co_daemon_run(recv)	6	co_win32_daemon_read_received	106 μ s
Total				649 μ s

**Figure 5.7:** Timing diagram for packets (cf. Table 5.1)

the ICMP PING response lies between checkpoints 3 and 4. This is relatively small compared to the total time from packet reception to packet sending on the Windows side. Thus the mere packet forwarding between Windows and the Linux VM dominates the total packet processing time, at least for packets as simple as ours. The explanation for this big delay is that each piece of network data is crossing memory space boundaries several times. Received packets are forwarded to the networking daemon, which copies them from kernel space to user space. After the Windows scheduler grants CPU time to the networking daemon, it forwards the data via a pipe to the coLinux daemon. When the coLinux daemon in turn gets hold of the CPU, it copies the data again and puts it back into kernel space via a `sysctl()` call. The `LINUX.SYS` driver, as recipient of the `sysctl()`, copies the data then into the passage page from where it is picked up by the Linux VM. Network packets sent by the Linux VM travel the same way back.

From this it is clear that the forwarding of network packets involves several context switches with associated scheduler interactions, and many copies of the data are made. This leads to inevitable waiting times. In addition, the whole handling chain is just too expensive in terms of CPU usage. When repeating the experiment on different systems, we have found that the network processing times depend on the speed of the installed processor, among other factors. For example, Table 5.2 on the next page shows the same measurement made on a somewhat slower PC with an 1.7 GHz Pentium III processor. The high CPU overhead also explains why it has not been possible to saturate the link with a data connection. Depending on the hardware, it was rarely possible to utilise more than half of the wire-speed on 100 Mb/s Ethernet.

Table 5.2: Current ping round-trip times with coLinux (unmodified)

Hardware	RTT
Pentium-M (1.4 GHz)	ca. 650 μ s
Pentium III (1.7 GHz)	ca. 950 μ s

5.4 Summary

The performance of a Linux VM realised with the current coLinux release has been benchmarked against a native Windows or Linux installation in terms of three key performance criteria: processor efficiency, memory access efficiency, and networking efficiency. While processor efficiency and memory access into coLinux' own memory partition produce quite comparable results, there is still something to do on the size of the memory partition and on both the latency and bandwidth of the network access.

To diminish the memory problem, the Linux VM must be enabled to access more memory directly. Several possible solutions are currently being discussed on the coLinux developers' mailing list [A⁺04]. Among others, there are: to use a physical memory defragmenter to allow a larger continuous piece of memory which can be allocated, or to allocate several disjointed regions of physical memory. Although each one is not easy to achieve, the coLinux developers team is working towards a solution here.

Regarding the network issues, things are a little more complicated. The current coLinux architecture is designed towards maximum flexibility, but happens to experience a performance loss. A major architectural change needs to be done here, in order to prevent network packets from crossing the kernel space/user space border. We are currently working on a new, high-performance networking solution which connects the LINUX.SYS driver directly with a network adapter driver.

Although much needs to be done, there are good chances that the outstanding performance problems can be solved. Together with the flexibility and functionality already found in a coLinux-based Linux VM, this will provide a firm foundation for the HARPY project.

6 Summary and Outlook

Since modern scientific and engineering research activities heavily rely on exact numerical calculations and simulations, the demand for computing power remains larger than the amount available. Many calculations consist of a huge number of comparatively small, independent jobs, e. g. to sweep all possible parameters for a design or to simulate a new workpiece in different applications. This scenario is known as high-throughput computing (HTC). HTC applications involving custom software are about as common as those using commercial solvers. Thus it is important to ease the development of new code but still be able to execute pre-compiled software.

To supply this need, research and development departments usually run some kind of compute server or cluster on which the actual calculations are performed. However, this cluster is typically overloaded and the budget for new hardware may be limited—whereas the typical office PC is sub-challenged with doing nearly nothing all day. The solution to this problem would be to use the whole available hardware, combining the cluster or compute server with idle office PCs. Middleware is needed to manage the increased complexity and to keep the handling of the execution environment out of the application code.

There are already some systems available to address this *opportunistic high-throughput computing* scenario, but they all lack some desirable features. Some demand a too high administrative overhead, while others require programs to be linked against special libraries. Therefore it is impossible to use commercial software such as CPLEX or Matlab.

HARPY is meant to fulfil our goals with as few disadvantages as possible in our field of application. It can be divided into four key components:

- The master execution nodes form the core of a HARPY pool. They are the original cluster of dedicated Linux machines.
- The client part is a screen-saver-like program running on non-dedicated nodes (office PCs running Windows) and enables them to join the cluster.
- The configuration server handles node addition and node eviction.
- The DRM system schedules the users' jobs on the pool in order to avoid overload situations.

We provide a requirements analysis and enumerate several design options for HARPY. Basically, there are three possibilities to execute jobs on the client side. Native execution would be the fastest, but requires specially targeted application versions for each platform. Sandboxing via a virtual machine (VM) can be done in a very slick way while hiding heterogeneity from the user application. Specialised client programs require a huge investment of labour, since they require the developer to rebuild all the middleware functionality by himself. Therefore we stick to the VM approach. Now that we have a number of machines running the same platform, there is a mechanism needed to distribute the jobs. Due to the dynamic nature of a HARPY pool which allows for dynamic node addition and eviction as client PCs become idle, application processes need to be moved between different nodes without restarting them. Thus a migration facility needs to be integrated into both the clients and master execution nodes. This service is provided by a single system image middleware. After an examination of several options, we choose openMosix to start.

We then present a more detailed design specification of the client. The Linux VM is constructed by the integration of coLinux, a user-mode Linux implementation, with the migration facility. Besides the virtual machine, the client also needs a configuration agent, handling configuration and setup issues during startup and shutdown, and a mechanism to perform remote software updates.

Crucial for the HARPY pool's overall performance is, besides global scheduling, the efficient execution on the client side. We therefore present a basic performance analysis consisting of several important micro-benchmarks. We measure how good the CPU, memory, and network utilisation is. We find out that by now coLinux makes very good use of the CPU, while some aspects about memory and network performance need to be reworked.

To conclude, HARPY is still in an alpha stage; several tasks have to be accomplished to make it a convenient end user system. On the client side the network link and memory management have to be improved. The configuration server is not yet implemented and there is still some work to be done with the DRM interface.

Nevertheless HARPY has quite a potential. Its design aims specifically at HTC scenarios and it is one of the first projects that cater both for users who roll their own application code and those who use commercial, packaged software. It features a plain, well known development environment and is able to reach near native execution speed. It can utilise node capacities dynamically and share the load among them. While leveraging an innovative and powerful backend, it sticks to standard DRM interfaces at the frontend, thus being able to benefit from the continued integration of resource management systems into Grid infrastructures.

Acknowledgements

We wish to thank the following persons for comments, hints, helpful discussions, suggestions, and code: Dan Aloni, Dietmar Fey, Reza Golgoon, Bruce Knox, Marcus Komann, Ian Latter, Ilkka Ollakka, and Mathias Rechenberg.

Bibliography

- [A⁺04] ALONI, DAN and OTHERS: *coLinux home page*. Accessed on 12th November, 2004. <http://www.colinux.org/>.
- [Alo04] ALONI, DAN: *Cooperative Linux*. In *Proc. Ottawa Linux Symposium*, Ottawa, Canada, July 21th–24th 2004. <http://www.colinux.org/publications/Reprint-Aloni-OLS2004.pdf>.
- [B⁺04] BAR, MOSHE and OTHERS: *openMosix project home page*. Accessed on 26th October, 2004. <http://www.openmosix.org>.
- [BDF⁺03] BARHAM, PAUL, BORIS DRAGOVIC, KEIR FRASER, STEVEN HAND, TIM HARRIS, ALEX HO, ROLF NEUGEBAUER, IAN PRATT, and ANDREW WARFIELD: *Xen and the art of virtualization*. In *SOSP '03: Proceedings of the 19th ACM symposium on Operating systems principles*, pages 164–177, Bolton Landing, NY, USA, 2003. ACM Press.
- [Bel04] BELLARD, FABRICE: *QEMU CPU emulator home page*. Accessed on 26th October, 2004. <http://fabrice.bellard.free.fr/qemu/>.
- [BGW93] BARAK, AMNON, SHAI GUDAY, and RICHARD G. WHEELER: *The Mosix Distributed Operating System*. Lecture Notes in Computer Science. Springer, Heidelberg, 1993.
- [BL98] BARAK, AMNON and OREN LA'ADAN: *The MOSIX multicomputer operating system for high performance cluster computing*. *Future Generation Computer Systems*, 13(4–5):361–372, 1998.
- [Buy99] BUYYA, RAJKUMAR (editor): *High Performance Cluster Computing*, volume 1. Prentice-Hall, Upper Saddle River, New Jersey, 1999.
- [Con05] CONDOR TEAM: *Condor Version 6.6.8 Manual*. University of Wisconsin–Madison, 2005. <http://www.cs.wisc.edu/condor/manual/>.
- [Dik01] DIKE, JEFF: *User-mode Linux*. In *Proc. Ottawa Linux Symposium*, Ottawa, Canada, July 25th–28th 2001. <http://user-mode-linux.sourceforge.net/slides/ols2001/>.
- [Dis04] DISTRIBUTED COMPUTING TECHNOLOGIES: *distributed.net project “Bovine” RC5*. Accessed on 18th November, 2004. <http://www.distributed.net/rc5/>.
- [DPG98] DOUGLAS P. GHORMLEY, ET AL: *GLUnix: A Global Layer Unix for a network of workstations*. *Software—Practice and Experience*, 28(9):929–961, 1998.

Bibliography

- [EMTT03] ESPOSITO, R., P. MASTROSERIO, G. TORTONE, and F. M. TAURINO: *Evaluating current processors performance and machines stability*. In *Proc. Computing in High Energy and Nuclear Physics*, La Jolla, CA, 24–28 March 2003.
- [FKK04] FEY, DIETMAR, MARCUS KOMANN, and CHRISTIAN KAUHAUS: *A framework for optimising parameter studies on a cluster computer by the example of micro-system design*. In KRANZLMÜLLER, DIETER, PETER KACZUK, and JACK DONGARRA (editors): *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number 3241 in LNCS, pages 436–441, Berlin, Heidelberg, New York, 2004. Springer.
- [GSFS04] GOECKELMANN, R., M. SCHOETTNER, S. FRENZ, and P. SCHULTHESS: *Plurix, a distributed operating system extending the single system image concept*. In *Proceedings of the IEEE Canadian CCECE/CCGEI Conference*, Niagara Falls, Canada, May 2004. http://www.plurix.de/publications/2004/CCECE04_final.pdf.
- [Hen02] HENDRIKS, ERIK: *BProc: The Beowulf distributed process space*. In *Proceedings of the 16th Annual ACM International Conference on Supercomputing (ICS'02)*, June 22–26 2002.
- [Jon93] JONES, MICHAEL B.: *Interposition agents: Transparently interposing user code at the system interface*. In *Symposium on Operating Systems Principles*, pages 80–93, 1993.
- [KWA⁺01] KORPELA, ERIC, DAN WERTHIMER, DAVID ANDERSON, JEFF COBB, and MATT LEBOSKY: *Seti@home—massively distributed computing for SETI*. *Computing in Science and Engineering*, pages 78–83, January 2001. <http://www.computer.org/cise/articles/seti.htm>.
- [MF99] MCGRAW, GARY and ED FELTEN: *Securing Java*. John Wiley & Sons, 1999. <http://www.securingsjava.com/>.
- [Mic04] MICROSOFT CORPORATION: *Microsoft Virtual PC 2004*. Accessed on 16th December, 2004. <http://www.microsoft.com/windows/virtualpc/>.
- [Mil04] MILLER, ZACH: *Condor administration tutorial*. Held at the UK Condor Week, October 2004. <http://www.nesc.ac.uk/esi/events/438/>.
- [MM01] MAURO, JIM and RICHARD MCDUGALL: *Solaris Internals: Core Kernel Architecture*. Sun Microsystems Press, Palo Alto, CA, 2001.
- [POV04] POV-TEAM: *POV-Ray Reference Manual*, 2004. For POV-Ray Version 3.6.1. <http://www.povray.org/download/>.
- [Ray96] RAYMOND, ERIC S. (editor): *The New Hacker's Dictionary*. MIT Press, Cambridge, MA, 3rd edition, October 1996.
- [SS05] SANTOSA, MULYADI and ANDREAS SCHÄFER: *Build a heterogeneous cluster with coLinux and openMosix*, February 2005. <http://www.ibm.com/developerworks/linux/library/l-colinux/>.
- [SSB⁺95] STERLING, T., D. SAVARESE, D. J. BECKER, J. E. DORBAND, U. A. RANAWAKE, and C. V. PACKER: *BEOWULF: A parallel workstation for scientific computation*. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

- [ST⁺04] SLIEPEN, GUUS, IVO TIMMERMANS, and OTHERS: *Tinc VPN home page*. Accessed on 27th October, 2004. <http://www.tinc-vpn.org/>.
- [Sun02] SUN MICROSYSTEMS INC., Santa Clara, CA: *SunTM ONE Grid Engine Administration and User's Guide*, 2002. <http://gridengine.sunsource.net>.
- [Sun04] SUN MICROSYSTEMS INC., Santa Clara, CA: *N1 Grid Engine Administration Guide*, 2004. <http://docs.sun.com/db/doc/817-5677>.
- [TL03] THAIN, DOUGLAS and MIRON LIVNY: *Parrot: Transparent user-level middleware for data-intensive computing*. In *Workshop on Adaptive Grid Middleware*, New Orleans, Louisiana, September 2003. <http://www.cs.wisc.edu/condor/publications>.
- [TTL04] THAIN, DOUGLAS, TODD TANNENBAUM, and MIRON LIVNY: *Distributed computing in practice: The Condor experience*. *Concurrency and Computation: Practice and Experience*, 2004. <http://www.cs.wisc.edu/condor/publications/>.
- [Van04] VANDERSMISSEN, WIM: *ClusterKnoppix home page*. Accessed on 26th October, 2004. <http://bofh.be/clusterknoppix/>.
- [Ver00] VERIDIAN SYSTEMS, Mountain View, CA: *Portable Batch System Administrator Guide, Release: OpenPBS 2.3*, 2000. <http://www.openpbs.org>.
- [VLR⁺03] VALLÉE, GEOFFROY, RENAUD LOTTIAUX, LOUIS RILLING, JEAN-YVES BERTHOU, IVAN DUTKA-MALHEN, and CHRISTINE MORIN: *A case for single system image cluster operating systems: Kerrighed approach*. *Parallel Processing Letters*, 13(2), June 2003.
- [VMw04] VMWARE, INC.: *VMware is virtual infrastructure*. Accessed on 16th December, 2004. <http://www.vmware.com/>.
- [W⁺04] WALKER, BRUCE and OTHERS: *OpenSSI home page*. Accessed on 26th October, 2004. <http://openssi.org/>.
- [Wri01] WRIGHT, DEREK: *Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with Condor*. In *Proc. Conference on Linux Clusters: The HPC Revolution*, Champaign, Urbana, IL, June 2001. <http://www.cs.wisc.edu/condor/publications/>.
- [WSG02] WHITAKER, A., M. SHAW, and S. D. GRIBBLE: *Scale and performance in the Denali isolation kernel*. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.
- [Y⁺04] YONAN, JAMES and OTHERS: *OpenVPN home page*. Accessed on 27th October, 2004. <http://openvpn.sourceforge.net/>.