Towards Performance-Driven System Support for Distributed Computing in Clustered Environments*

John Cruz† Kihong Park‡
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
{cruz,park}@cs.purdue.edu

CSD-TR 98-035
October 27, 1998

Abstract

With the proliferation of networked distributed resources and the prevalence of workstation clusters as a dominant computing platform, providing adequate system support for distributed computing, including parallel computing, has become an important problem. A principal focus of previous works has been on enabling technologies that facilitate various forms of transparency including interoperability and ease-of-programming. Less emphasis has been given to performance and efficiency considerations, the dual, but equally important side to effectively achieving distributed computing in networked environments.

This paper describes a set of performance features, their properties, implementation, and evaluation in a software support environment called DUNES. The main performance features consist of push/pull-based forms of active and passive end-point caching, communication-sensitive load balancing, distributed demand-based paging, and adaptive communication control. Although each feature targets a separate aspect of performance, collectively, they affect the scheduling of distributed resources to application processes where both communication and computation requirements are taken into account.

The architecture of DUNES—in addition to incorporating the aforementioned performance features—allows commodity operating systems to be easily transformed into a distributed operating system while achieving complete transparency with respect to the existing application base as well as preserving semantic correctness. We show performance measurements of a Solaris UNIX based implementation of DUNES on Sparc and x86 architectures over LAN environments. We show that significant performance gains in terms of parallel application speed-up and high system throughput is achievable.

---

*Supported in part by NSF grant ESS-9806741.
†Additionally supported by a fellowship from the Purdue Research Foundation (PRF).
‡Contact author; tel.: (765) 494-7821, fax.: (765) 494-0739. Additionally supported by NSF grant NCR-9714707, and grants from PRF and Sprint.

1 Introduction

1.1 Motivation

With the advent of high-speed networks connecting a large number of high-performance workstations via local area and wide area networks, harnessing their collective power for distributed computing, including parallel computing, has become a viable goal. Concurrent applications can span from everyday applications to numerical and supercomputing applications which include a diverse spectrum of computational problems ranging from partial differential equations to global weather simulation to molecular sequence analysis [3, 5, 19, 28, 52, 62, 72, 73, 33, 74].

In addition to intrinsic limitations such as latencies introduced by increased physical distances between networked hosts, two key issues need to be addressed to facilitate a distributed computing environment capable of emulating the prowess of tightly coupled parallel computers—communication control and load balancing. Although these issues arise in parallel machines as well, their impact is amplified in workstation networks requiring new solutions.

With respect to communication control, the lack of special calibrated communication facilities in the form of interconnection networks renders a workstation cluster more prone to congestion effects when large volumes of data are transferred between hosts. In the case of load balancing, balancing of processor load without proper regard for communication costs can deteriorate performance when network communication becomes a dominant factor. Furthermore, uneven access to network bandwidth among hosts in the system increases the variance in computational progress of parallel applications distributed across participating hosts. This, in turn, aggravates synchronization penalty stalling application progress.

A myriad of software support environments have been advanced in the past with a view toward facilitating concurrent applications in workstation environments [2, 6, 11, 12, 16, 17, 23, 35, 51, 67]. A principal focus of previous works has been on *enabling technologies* that achieve various forms of transparency including interoperability and ease-of-programming. On the performance side, significant work has been done in load balancing and process migration [4, 7, 14, 22, 26, 27, 30, 39, 46, 50, 61, 63, 76], a key component to achieving parallel speed-up and high system throughput.

More recently, performance studies of LAN- and WAN-based systems have shown the importance of controlling network communication for improving parallel or distributed application performance [15, 21, 42, 44, 45, 56, 66]. The sensitivity of application performance to congestion effects is directly dependent upon the communication/computation ratio and degree of synchrony. An application with a high communication/computation ratio is prone to generate periods of concentrated congestion which leads to debilitating communication bottlenecks. Moreover, if two or more such tightly coupled processes stemming from communication-intensive applications are split apart and scheduled on separate hosts, then the resulting communication overhead can overshadow any gain obtained from a more balanced load.

Synchronous applications—in particular, those with lock-step computation-communication iterations—are subject to synchronization penalties which are determined by the progress rate of the *slowest* processing element. The latter can be shown to be "exponentially sensitive" in the number of nodes or workstations participating in the computation [43].

1.2 Problem Statement

This paper addresses the issue of achieving parallel speed-up of concurrent applications and high system throughput in shared network environments, where scheduling of processes to resources is determined by an *integrated* approach to computation and communication control.

A principal lesson learned from load balancing is that, in the case of dynamic load balancing, processes best suited for migration are those that are largely independent (or isolated), long-lived, and small in size. When this is not the case, the gain obtained from a more balanced load can be outweighed by the resulting amplification of communication cost—single host interprocess communication or file access is turned into its more erratic and expensive cousin, network communication—as well as the overhead associated with process migration itself. This, in turn, has lead to the practice of static load balancing, both for its simplicity and the difficulty of performing cost/benefit analysis of dynamical load balancing at run time.

The latter approach is satisfactory if a distributed application is executed in a dedicated, closed environment and application-specific information obtained using a priori structural information or at compile time is sufficient to affect a distribution of load that minimizes application completion time. However, it has been shown that dynamic communication control in the form of application-sensitive congestion control is needed to achieve stable network behavior and high throughput which can reduce application completion time by several factors [42, 44].

When the environment is nonstationary, as is increasingly the case in today's shared workstation networks whose underutilized resources we seek to harness, with new applications joining and departing continuously and external factors including computations and traffic flows triggered by Web browsers and other applications exerting a nonnegligible influence on the contention level of shared resources, a dynamic resource allocation scheme is needed to achieve both high application performance and system throughput.

The dynamic scheduling of distributed resources to application processes must explicitly take into account both computation and communication requirements and their costs, and it must be responsive to nonstationary changes in system state which may bring forth opportunities—and, in some cases, necessitate remedial actions—to improve or preserve performance. To achieve these goals, one, mechanisms must be put into place that facilitate the efficient use of underutilized resources, two, the state of the system must be accurately and efficiently monitored on-line to provide reliable information for decision making, and three, algorithms are needed that make use of the mechanisms and state information in an appropriate way.

1.3 New Contributions

The contributions of this paper are twofold. First, we propose an integrated approach to computation and communication control where the scheduling of application processes to distributed resources is affected by explicit incorporation of both computation and communication requirement/cost considerations. This is achieved by devising a set of performance enhancement features aimed at reducing communication cost which, when combined with a facility for accurate and efficient monitoring of system state, enables a communication-sensitive load balancer to schedule resources effectively.

Second, we implement our functional and performance features in a library distributed operating system called DUNES (**D**istributed **UN**ix **E**xten**S**ion) wherein the performance gains are demonstrated using controlled experiments[1]. Although the performance enhancement features, monitoring mechanisms, and integrated distributed scheduling algorithm are platform independent and thus portable to other environments, the architecture of DUNES provides additional features including efficiency, transparency, and deployability which makes its implementation and realization particularly attractive.

The principal components in the performance feature category are those that facilitate the reduction of communication cost when coupled processes are executed on separate hosts or otherwise separated from software resources such as files. Dynamic monitoring of system state involves maintaining a continuously updated profile of "who talks to whom and how much" (interprocess coupling), "how compute- or communication-intensive is a process" (communication-computation ratio), and "what is the utilization of hardware resources" (CPU and bandwidth utilization). The control algorithm—making use of state information encompassing both computation and communication behavior—dynamically schedules application processes so as to enhance performance.

Performance Features As part of the distributed OS functionality, our system implements dynamic process migration following the user-level mechanism employed in Condor [51]. However, unlike in Condor, our dynamic process migration mechanism handles *dependencies* arising from interprocess communication and file access maintaining transparent bindings consistent with UNIX semantics. Our process migration facility also supports dynamic process creation using `fork` and related activities (e.g., `exec`), again, providing functionality consistent with standard single processor UNIX semantics[2].

To offset or hide overhead stemming from the expanded distributed OS functionality, in particular, those due to dependencies arising from interprocess communication and file access in the presence of process migration, we implement a set of performance enhancement mechanisms de-

[1] Some researchers may object to calling a non-kernel-based implementation of distributed operating system functionality a "distributed operating system." We explain our reasons below.

[2] What we mean by "consistent with standard single processor UNIX semantics" is that, fixing a specific version of UNIX, the execution obtained from our system is sequentially consistent with the corresponding single processor system.

scribed below:

- *Active end-point caching* To hide the (network) communication latency incurred by processes engaging in IPC that have been split apart due to migration, we employ a prefetching or push-based caching mechanism which forwards data written to a communication channel to the target process without waiting for the issuance of reads. Since network latencies can be large—especially when the network is congested—this mechanism attempts to maximize the benefit of concurrency by hiding its potential communication cost.

- *Passive end-point caching* Similarly to active end-point caching, we seek to minimize the cost of remote access to files by a process which has been separated due to migration. We employ a form of prefetching coupled with paging and "client side" caching such that reads and writes can be handled locally at the remote host whenever possible. We implement a cache consistency mechanism with single writer/multiple reader semantics which conforms to standard UNIX semantics.

- *Demand paging of process image* We seek to reduce the cost of migration in which a major portion of the cost is due to the transfer of an entire checkpointed image. Instead of sending an entire checkpointed image to the host where a process is to be resumed, we send only those pages that correspond to the current working set, and the remaining pages are fetched on demand. This ensures that the time taken for migration is considerably less when compared against the time incurred transferring the entire checkpointed image.

- *Communication-sensitive load balancing* Whereas the aforementioned mechanisms try to minimize the cost of facilitating dependencies over a distance, communication-sensitive load balancing tries to prevent strongly coupled processes (coupled with other processes or files) from being split apart in the first place if the benefit of parallelism is deemed less than its cost. This is enabled by an efficient run-time state monitoring mechanism that quantitatively estimates process-to-process and process-to-file communication patterns which can then be used to perform a form of cost/benefit analysis to avoid unfruitful migrations and instantiate fruitful ones.

Another important performance feature is *adaptive communication control* where application-sensitive congestion control is applied to improve the effective throughput achieved by processes belonging to a concurrent application distributed across a network. In previous works [42, 44, 45] we have shown that significant performance gains—up to a factor of 4 under heavily congested network conditions—can be obtained for parallel applications if a state-of-the-art dynamic congestion control [58] is applied to regulate traffic flow on a per-application basis. We omit adaptive communication related results due to redundancy and brevity reasons.

Architectural and Functional Features Another goal of the paper is to report our experience with designing, implementing, and evaluating a user-level "off-the-shelf" distributed operating system[3]—DUNES—that, in addition to incorporating the aforementioned performance enhancement features is efficient, easily deployable, transparent, and extensible. The architectural features, individually, have limited appeal. However, when combined, they coalesce into an approach to distributed operating system design whose properties render it practically viable. We elaborate on each of the features below:

- *Off-the-shelf* Our system is "off-the-shelf" in the sense that it runs on top of commodity operating systems imparting distributed OS functionalities while respecting the existing application base. As a by-product, it is easily deployable and portable in the sense that the same architecture can be fitted on different commodity operating systems (e.g., UNIX, Windows NT) and hardware platforms.

- *Transparent* The first form of transparency is a side effect of the off-the-shelf feature in that existing applications need not be recompiled—just *relinked* with a modified system call library—to run on DUNES. This significantly reduces the barrier faced by new operating systems and computing environments being adopted by existing systems.

  The second form of transparency is *functional transparency* where DUNES ensures that various forms of dependencies including process-to-process and process-to-file depencies are transparently maintained by the system in the presence of dynamic scheduling. Thus, for example, if a process migrates to another host, its existing dependencies continue to be preserved completely transparent to the process.

  The third form of transparency is *semantic transparency* where, in the process of achieving functional transparency, the semantic correctness of application execution is ensured. DUNES provides a complete *single system image* to the user which extends to semantic correctness: a concurrent application running under DUNES across multiple workstations achieves a sequentially consistent execution as its counterpart on a single processor host. In particular, DUNES preserves single processor UNIX semantics.

- *Efficient* Our system implements distributed OS functionality with minimal overhead by implementing the functionalities as a thin layer above the system call layer, i.e., the interface to the set of services exported by a kernel. If kernel modification—thus violating both the off-the-shelf and transparency features—is not an issue, the same modifications can be implemented inside the kernel, however, yielding no performance difference except that the added instructions would be executed in kernel mode.

- *Extensible* Our system is easily extensible with respect to its functionality including different conflict resolution schemes ("who gets what" policy), real-time scheduling for multimedia

[3]It is, at the same time, a software system support environment in the sense that all the features are implemented as user-level libraries, albeit, at a "low level" as explained below.

tasks and other QoS-sensitive applications, and run-time system state monitoring. The latter is used to facilitate dynamic communication-sensitive load balancing. Extensibility is a by-product of DUNES' library operating system approach to imparting new or extended functionality.

Our goals are ambitious in the sense that we seek the "best of both worlds" by designing a distributed operating system that is easily deployable using the technique of *library operating systems* to impart distributed OS functionality while delivering significant performance improvements which approach that of kernel-based distributed operating systems. Distributed OS functionality is injected into a commodity OS—our implementation is in the context of UNIX (SunOS 5.5.1) on both Sparc and x86 architectures—by redefining the service access point or system call interface (mostly wrapper code trapping to `syscall` in Solaris) to kernel services, replacing the system call library with our modified library and relinking applications with the new library. Thus applications need not be recompiled. This method of adding functionalities has been used in the past, among other things, to facilitate user-level checkpointing and process migration in UNIX [51, 53], and it can be viewed as part of the general framework of library operating systems [47] to extending OS functionality.

In the context of concurrent application development for parallel and distributed applications, the *programming model* that DUNES exports to the programmer is one of writing concurrent programs for a single processor UNIX environment. If the concurrent application is correctly written for a single processor environment, then DUNES guarantees that it will execute correctly in the distributed resource environment. If the granularity of an application's concurrency is variable and thus controllable by the programmer, then, as with parallel architectures, sufficient granularity needs to be imparted such that parallelism—if beneficial—can be exploited over a workstation network environment. This can also be affected with the assistance of parallel compilation tools that transform a serial program—oftentimes annotated—into a concurrent form suitable for parallel execution. If DUNES' communication-sensitive load balancer does not deem beneficial to distribute load at the granularity allowed by the concurrency of the application, then, as with parallel computers, multiple application processes are scheduled on a single host.

The rest of the paper is organized as follows. In the next section we summarize related work. This is followed by Section 3 which describes the basic DUNES architecture including its functional and performance features. In Section 4 we describe communication-sensitive load balancing where all of the performance-oriented features are brought together to affect integrated computation/communication control. Section 5 shows performance results of a DUNES implementation for Solaris UNIX (SunOS 5.5.1) on both Sparc and x86 architectures measured over private, controlled LAN-based workstation network environments. We conclude with a discussion of our results and future work.

2 Related Work

There are many distributed computing platforms and distributed operating systems currently in existence [8, 20, 24, 48, 51, 55, 68, 69]. The former are spearheaded by recent developments in network computing [68, 70, 71] where the primary focus has been on enabling technologies that achieve platform independence and allow heterogeneous distributed resources to be harnessed across networked environments. Distributed operating systems, for the most part, are written from scratch and are kernel-based: distributed OS functionalities are resident inside the kernel. Building a complete distributed operating system from scratch is a monumental task. A subarea of operating systems—extensible operating systems [9, 29, 32]—tries to make the building of new functionalities and reuse of existing functionalities easier through various means. In all cases, there is a well-defined interface to kernel services and other predefined services.

The microkernel approach to operating system design tries to make the functionality exported by a kernel minimal with behavioral customizations carried out at the user level. The exokernel [29] is an extreme instance where management functionality is separated from protection functionality and only the latter is provided by the kernel. Thus memory management, scheduling, and other traditionally kernel-based services are now implemented at the user-level allowing for maximum user control and flexibility. It is clear that a complete distributed OS can be built at the user level on top of Xok—the exokernel for x86-based machines—and exported as a set of libraries, also called library operating systems (libOS) [47]. A key question associated with the microkernel approach is whether the resulting systems perform as efficiently as their monolithic brethren (e.g., UNIX).

The library OS approach to imparting resource management functionalities, to some extent, blurs the dividing line between "hard core" distributed operating systems of the past—by definition, kernel-based—and the abundance of network computing platforms today. In the libOS approach, resource management functionalities—including some only previously found inside the kernel even for microkernels—are purposely implemented using user-level libraries. Whether the resulting system is called a software support environment or distributed operating system is a matter of taste and interpretation. Our approach to imparting distributed OS functionality to commodity operating systems can be viewed as an instance of libOS, albeit interfacing with a monolithic kernel rather than a microkernel. We seek the best of both worlds—transparency from network computing and efficiency from distributed operating systems. The most challenging technical (i.e., mechanistic) aspect of distributed OS functionality—process migration—can be achieved at the user-level on top of commodity UNIX [51, 53, 60] and forms the starting point for facilitating full distributed OS functionality.

Some kernel-based process migration facilities include those in Amoeba [69], Clouds [24], V [20], MOSIX [8], and Charlotte [31], to mention a few. The Tui system [65] is interesting from the perspective that it supports process migration across heterogeneous machines. Process migration for heterogeneous environments is also studied in [13]. However, due to the significant translation cost involved, it is rarely considered a viable dynamic load balancing strategy. A survey of process migration mechanisms and related issues can be found in [64].

Our system is similar to Condor [51] in that it follows the latter's user-level process migration scheme. However, unlike Condor, our dynamic process migration mechanism handles *dependencies* arising from interprocess communication, network communication, process creation, and file access while maintaining transparent bindings consistent with UNIX semantics. Condor is restricted to migrating "stand-alone" processes with support for remote files access using RPC. Our process migration facility supports dynamic process creation using `fork` and related activities (e.g., `exec`), and thus increases the set of applications that can be supported. This is important from the perspective that present day applications tend to engage in some form of interaction—frequent or infrequent—with other applications and resources, and thus maintaining dependencies correctly with respect to standard UNIX semantics is an important requirement.

Another related system is GLUnix [37]. However, GLUnix does not support process migration and dynamic load balancing, and it does not possess the performance enhancement features of DUNES. A similar observation holds for PVM (Parallel Virtual Machine) [68], an execution environment for the development and execution of large concurrent and parallel applications that consist of many interacting, but relatively independent, components. MPVM [18] and DynamicPVM [25] are extensions to PVM that support process migration. They follow the approach used by Condor to checkpoint and restart processes.

The benefit of prefetching and caching has a long history spanning a number of areas from database systems, to file systems, to computer architecture and operating systems, and has gained renewed interest due to the proliferation of networked systems where latency hiding has become imperative for performance [1, 38, 59]. Sprite [54], for example, adopts non-write-through file caching employing a simple cache consistency mechanism and it is shown that "client side" caching—corresponding to our passive end-point caching—can reduce both server and network load significantly. The DFS file system of OSF's Distributed Computing Environment (DCE) implements a token-based cache consistency mechanism with expiration timers which effectively yields a single writer/multiple reader semantics which, in turn, conforms with UNIX semantics. Our passive end-point caching mechanism adopts a token-based cache consistency mechanism similar to DFS' scheme. This is motivated by the relative simplicity of implementing single writer/multiple reader semantics. A recent study by Guy *et al.* [38] has shown that an optimistic replication policy based on the former leads to satisfactory performance and provides room for further improvement.

3 Architecture of DUNES

3.1 Overall Structure

DUNES (**D**istributed **UN**ix **E**xten**S**ion) is a distributed operating system designed using the approach of *library operating systems* [29]. Operating systems export a number of services executed by the kernel via the interface of systems calls, e.g., in the case of UNIX, the system call stubs in

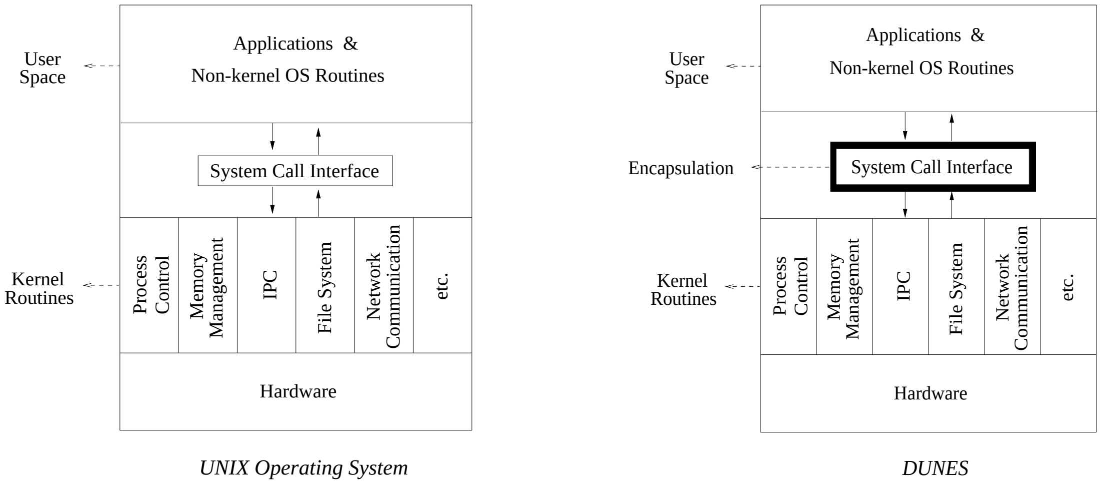


Figure 3.1: Left: Structure of typical UNIX operating system. Right: UNIX operating system augmented by system call encapsulation and DUNES resource management routines.

the standard C library. Since all kernel services must be accessed through this narrow, well-defined corridor, by modifying the system call interface new functionalities can be added to the operating system without changing the kernel.

In addition to user-level extension of operating system functionality, this approach allows an existing application base to be run without recompilation by relinking with the modified library thus achieving backward compatibility. These features, in turn, are conducive to deployability in the sense that a stand-alone commodity OS can be turned into a distributed operating system just by installing a new library. Proximity and intimate control over system resources, short of changing the kernel, are preserved via a thin, transparent encapsulation layer which distinguishes this approach from network computing based approaches which are prone to introduce significantly more overhead. Transparency and deployability, we believe, represent an important advantage over kernel based distributed operating systems which, in spite of their numerous manifestations [8, 20, 24, 55] have as yet achieved only partial success at wide-spread use due to the practical burden of incompatibility and inertia posed by stand-alone commodity operating systems. Figure 3.1 illustrates this design methodology.

The library OS approach to designing distributed operating systems can be applied to both microkernels and monolithic kernels where in one extreme instance of the former—MIT's exokernel [29]—except for protection functionality, all other resource management functionalities including memory management are performed by user-level library routines. This degree of customization allows the potential for increased efficiency, an active area of operating systems research. As we seek the "best of both worlds"—transparency and efficiency—in the case of monolithic kernels (e.g., UNIX), to approach the efficiency level of microkernel based designs and kernel based distributed operating systems, we implement a set of performance enhancement features and resource control mechanisms aimed at hiding intrinsic efficiency limitations and thus improving performance.

3.2 DUNES Components

In this section, we give an overview of DUNES' architectural components which consist of functional and performance features. The functional components form the basic layer of enabling mechanisms that allow distributed resources across a workstation network to be transparently shared and the system state to be transparently and efficiently monitored. The performance enhancement components are features built on top of the functional components and they facilitate the *efficient* sharing of distributed resources through pull/push based caching of active and passive end-points, demand paging of process state, and integrated computation/communication control.

3.2.1 Functional Features

Transparent Processor Sharing As with other distributed operating systems and distributed computing environments, a principal component of DUNES is the transparent enabling of processor sharing across different workstations which facilitates increased system throughput and application performance if the associated overhead is not "too large." The primary enabling feature of transparent processor sharing is *process migration*. Techniques for transparent process migration using user-level libraries for checkpointing and restart are well-known [51, 53, 60] and we follow an analogous strategy in our own implementation.

First, application binaries, when being relinked with the modified system call library, are also linked with the DUNES start-up routine `Main()` which, after initialization, transfers control to the application binary proper by calling `main()`. DUNES' start-up routine essentially installs an "all-purpose" signal handler for the SIGUSR1 signal which encapsulates three separate functionalities: one, responding to the load balancer's command to checkpoint for subsequent process migration, two, to respond to `timer_create`'s alarms for periodic logging of run-time monitored short-term communication and computation information, and three, for checking if the signal originated from the user process itself, in which case, the user's signal disposition is invoked. The checkpointed image is then migrated by a process migration daemon to its counterpart on a designated destination host.

A migrating process maintains state information on the host where it was initially started—called the *home base*—in the form of a *proxy process* that subsequently handles its process-to-process and other dependencies transparently. This is done by `exec`ing the proxy code from inside the application process—the last action of the DUNES SIGUSR1 signal handler after checkpointing—which then inherits the relevant properties of the migrated application process. The migrated application process also maintains a corresponding *remote proxy* on the destination host which, in addition to acting as a liaison, also carries out other functionalities including managing the local cache for passive and active end-point caching. The home base proxy, by monitoring all open descriptors belonging to the migrated application process (using `select`) on the home base, is then

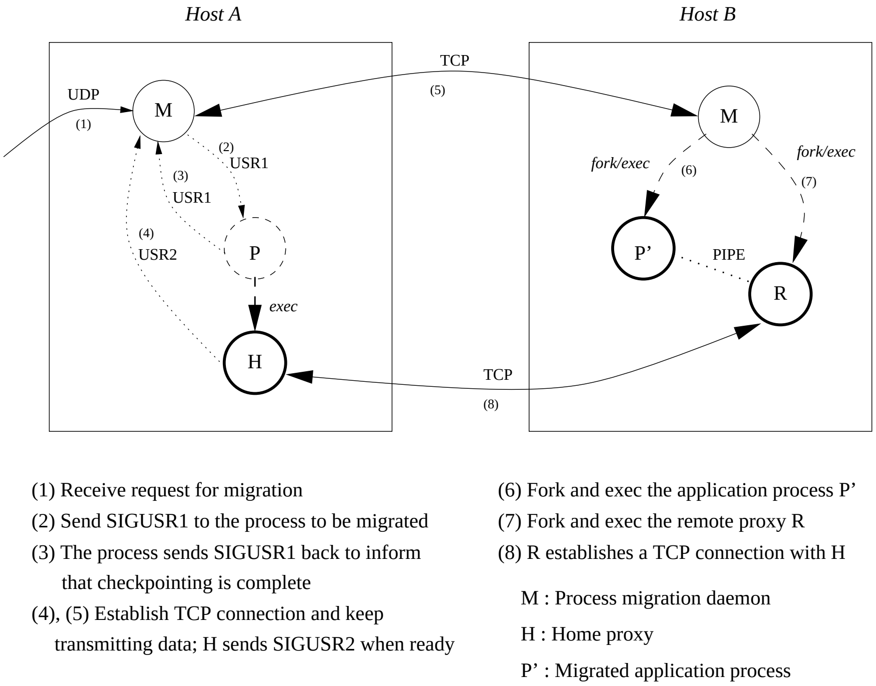


Figure 3.2: Migration of process P from host A to host B and resulting triangle relation between migrated process P' via remote proxy R to home proxy H.

able to transparently handle the dependency relations described below.

If a migrated process is subsequently migrated again, then all state information on the previous remote host is deleted and the configuration reached is indistinguishable from the previous migrated configuration: home proxy on the home base and remote proxy and migrated application process on the destination host. Thus repeated migration does not increase the complexity of the system state. A snapshot of the system—when restricted to the state information for a single migrated process—is shown in Figure 3.2. After migrating process P from host A to host B, what remains is the triangle relationship between migrated process P' via remote proxy R to home proxy H.

Transparent Dependency Maintenance Transparent process migration, for isolated processes, is a straightforward matter. All processes have some form of dependency (e.g., parent/child relation in UNIX), but more importantly, most processes engage in some form of activity such as file access, interprocess communication (IPC) on a single host, and network communication over separate hosts. Furthermore, a process, after migration, may fork off another process which can complicate the picture significantly. Transparently maintaining dependencies in the presence of mobility using user-level techniques is a nontrivial challenge. Condor [51], for example, supports transparent file access but does not allow process migration in the presence of IPC, network communication, or `fork`. One practical justification for this is that, other things being equal, the processes that benefit most from migration are isolated processes. Transparent file access, even for the most rudimentary processes, is a necessity and this is provided through various means including interfacing to a network file system (if one exists).

However, more and more, processes engage in some level of IPC and network communication—for some applications such as parallel computing applications the communication/computation ratio can be exceedingly high—and excluding them from dynamic load balancing may incur a significant opportunity cost [40, 41]. DUNES provides the mechanisms for maintaining dependencies transparently, and the issue of whether for certain processes it would be beneficial to migrate is left to a performance enhancement feature—the communication-sensitive load balancer—to decide. In this way, we have the option of engaging in migration when it is beneficial to do so even in the presence of nontrivial interprocess coupling (cf. the sections on performance measurements), and refraining from doing so if it is deemed detrimental.

Another important aspect of maintaining dependencies transparently is the issue of correctness and semantics. If transparency is "provided" but program execution correctness—according to some fixed criterion—is not preserved, then the resulting system can be potentially perilous, burdening the programmer with additional concerns. DUNES' functional features provide single processor UNIX semantics which, in turn, is based on the notion of sequential consistency. The programmer can write concurrent code with single processor UNIX semantics in mind and the transparent dependency mechanism will guarantee that its execution will be sequentially consistent with the program's execution on a single processor UNIX system.

When a process P migrates from a host, it leaves a proxy process H on that machine. On the destination machine a proxy process R is created (see Figure 3.2). The application process talks to R through a pipe and R talks to H over the network using TCP. For every system call that a migrated application invokes, a request is sent to R which is forwarded to H. H executes the syscall call on behalf of the application process and sends the result back to the application process through R. As H is `exec`ed by the application process, it inherits file descriptors, signal dispositions, and other relevant properties leaving the dependencies intact for transparent maintenance.

When a migrated process migrates again, R on the current host is terminated and started on the new host. This ensures that there is no dependency left on the host on which a migrated process was previously executing. When a migrated process `fork`s, a similar structure (H and R) is created for the child process. This ensures that the child process can be separated from the parent process for further migration. Subsequently, except for the parent-child relationship and all that it entails, the child is an autonomous entity and the resulting configuration is indistinguishable from one where the child process would have been `fork`ed first—on the home base—and then migrated. In other words, the two operations *commute*.

Communication and Computation Monitoring The library OS approach to distributed operating system design has the beneficial side effect that run-time monitoring of communication activities can be done transparently, accurately, and efficiently. Since all process-to-process

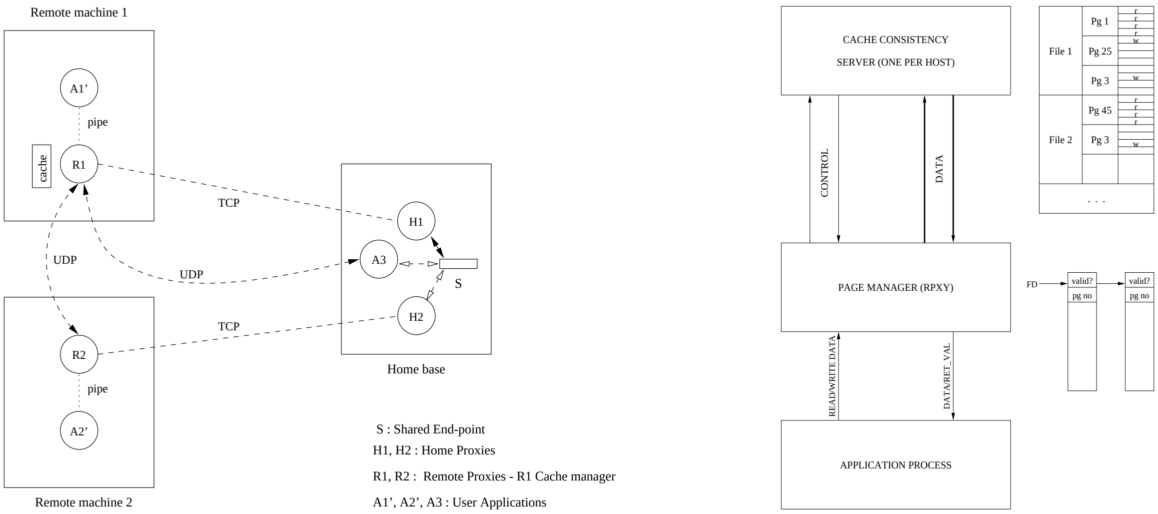


Figure 3.3: Left: Active end-point caching. Right: Passive end-point caching (the proxy process on the destination machine RPXY acts as the page manager).

or process-to-file communication must go through system calls[4], by implementing simple counting mechanisms inside `read`, `write`, `send`, `receive`, and other I/O related system calls on a per descriptor basis, the communication behavior of application processes can be readily monitored. Computation information such as CPU utilization on a per process basis is maintained by the kernel (e.g., `/proc` file system for Solaris) and can be periodically queried to obtain both long-term and short-term behavioral information. In DUNES, run-time communication and computation monitoring is used by the communication-sensitive load balancer to affect distributed scheduling based on an integrated approach to computation and communication control.

3.2.2 Performance Features

Active End-Point Caching When communicating processes on a single host are split apart onto separate hosts or processes engaging in network communication are migrated to more "distant" hosts—either physically (link latency and physical bandwidth) or logically (queuing effects and available bandwidth)—then even though the resulting action may yield a net gain in system throughput and application completion time, the performance benefit can be further improved if the effective communication cost is reduced by employing a form of push-based caching. For example, in the case of `fifo` or `pipe` based IPC turning into network communication due to process migration, whenever a `write` is executed, the data is immediately shipped to the reader (in the case of multiple readers to the most "likely" reader) such that when a reader executes a `read` operation,

[4]One exception is IPC through mapped shared memory segments although, if efficiency is not a consideration (shared memory is the fastest form of IPC and is used, by convention, with this in mind by application programmers), then even this can be done.

the data is already in the reader's local cache and access time is close to the cost of a local `read`. In the multiple reader case, some care has to be taken to prevent one reader (i.e., process) from being starved by another[5] as well as making sure that the cached data is shipped to the correct destination given that IPC and network communication data is of a consumable nature.

Active end-point caching not only hides communication latency but it also enables scheduling actions involving process migration to be fruitful when, without, the same actions may be determined to be detrimental. Thus this leads to further opportunities for performance improvement which would otherwise not be accessible. Of course, this assumes that a distributed scheduler is able to make use of run-time communication/computation information appropriately by performing accurate cost/benefit analysis of various possible actions. This issue is discussed in the design of communication-sensitive load balancing.

Figure 3.3 (Left) shows a typical scenario involving a read-shared `fifo` accessed by two or more migrated processes. One of the migrated processes' proxy takes charge as the cache manager. The other proxies contact the cache manager to get the cached data. When applications that run on the home base want to access data from the active end-point, they too have to contact the cache manager. This access subsequently disables caching to reduce the overhead for processes running on the home base.

Passive End-Point Caching Analogous to active end-point caching, passive end-point caching uses a push-based or prefetching mechanism to hide communication latency when files are accessed remotely by a process due to separation. Network file systems (e.g., Sun NFS) employ caching to reduce access times. DUNES, by default, does not assume the existence of a network file system to achieve generality—optimizations to interface with particular network file systems, if detected to be present, are in progress—but rather engages in its own push-based caching scheme. We use a page-based system with k (by default $k = 2$) pages and page size S (by default $S = 1$kB) with LRU page replacement policy (see Figure 3.3 (Right)).

DUNES implements single writer/multiple reader semantics using a cache consistency manager which is consistent with single processor UNIX semantics. The granularity of access is on a per-page basis, and as is well-known, there is a trade-off between page size and frequency of conflict, with increased granularity carrying a commensurate management overhead cost. Single writer/multiple reader semantics is the simplest but also most restrictive cache consistency protocol. However, previous studies on read/write access in UNIX file systems and more recent studies for collaborative workgroup environments [38] have shown that concurrent read/write access—and even more so for write/write access—to common files is a infrequent event which, by Amdahl's Law, warrants the use of more optimistic consistency protocols.

Demand Paging of Process Image When migrating a process, there are two costs involved—a fixed cost and a variable cost. The fixed cost consists of setting up the proxies and the variable cost

[5] This may still be correct with respect to single processor UNIX semantics but a programmer oftentimes relies on implicit performance assumptions that the timesharing class of UNIX schedulers provide.

depends on the size of the process image—in particular, its virtual address space—being migrated. In [26], different methods of transferring virtual memory are discussed. They apply to systems where process migration is supported by the kernel. One of the promising approaches (described in [26]) is the method used by Accent [75] called the *copy-on-reference* mechanism. Here, when a process migrates, only the process state is transferred. The migrated process begins execution almost immediately. Virtual memory pages are transferred only on demand, resulting in short delays during execution.

In DUNES, a mechanism similar to Accent's copy-on-reference mechanism is incorporated. When a process is checkpointed, the checkpoint image is written to disk and the virtual memory freed. Only the text segment along with the process state is transferred to the destination machine, and the other segments are transferred on demand. When a checkpointed process is restored, all mappings other than the mapping of the text segment are marked inaccessible. When an address within a segment marked inaccessible is accessed, a SIGSEGV signal is delivered to the application process. From the fault address, it is determined whether the address is valid or invalid. If the address is valid, the corresponding page is restored from the source machine's disk and is marked accessible. When control returns from the signal handler, the process continues execution from where it left off. Further optimizations, such as freezing the process on the source machine rather than writing the image to disk, are possible. These optimizations can further reduce the start-up time after migration.

Communication-Sensitive Load Balancing A principal lesson learned from load balancing is that, in the case of dynamic load balancing, processes best suited for migration are those that are largely independent, long-lived, and small in size. When this is not the case, the gain obtained from a more balanced load can be outweighed by the resulting amplification of communication cost as well as the overhead associated with process migration itself.

Critical to the success of communication-sensitive load balancing is a method for cost/benefit analysis that accurately estimates or predicts the "goodness" of the configuration reached after the execution of an action that may involve one or more process migrations. This, in turn, is dependent upon an effective measure of goodness. We define such a measure called *progress rate* which incorporates both communication and computation requirements—as exhibited by a process' behavior—which is related to the communication/computation ratio of a process. With the assistance of our run-time monitoring mechanism, we are able to predict the progress rate of a potential next configuration, and by comparing with the *measured* (or *observed*) progress rate of the current configuration, determine the ranking of candidate actions and decide whether it is worthwhile to take an action. The communication-sensitive load balancer—centralized or distributed—uses the predicted progress rate of candidate configurations and iteratively takes actions until no further performance improvement is deemed possible. The progress rate estimation procedure is accurate as long as actions involving process migrations are *nonoverlapping* and admits an efficient form of distributed control.

We note that one other important factor in the success of dynamic load balancing is the process lifetime distribution of application processes. *Given* the progress rates of two different configurations, there is a break-even point, parameterized by the cost of process migration itself, where only if the process' lifetime is longer than a certain period—which itself depends on progress rate—is process migration beneficial. Recent work [40, 41] in process lifetime distribution of UNIX processes has shown that process lifetimes are heavy-tailed (i.e., the tail decays hyperbolically, not exponentially) which allows effective prediction due to the presence of long-range correlation structure. Lifetime prediction is a separate subject matter unto itself of independent interest and is not part of the scope of this paper. In the remainder of the paper, we assume suitably long-lived processes where the estimation of predicted progress rate is the primary concern.

4 Communication-Sensitive Load Balancing

The following sections describe the various components of communication-sensitive load balancing including a definition of the resource allocation problem, progress rate, its estimation, and the overall structure of the integrated load balancer.

4.1 Integrated Resource Allocation Model

The hardware resources and their configuration are described by a graph structure called the resource relation graph. A similar graph structure is used to describe processes, their couplings which includes interprocess dependencies as well as dependencies on other software resources such as files.

First, let us define the *resource relation graph* or simply *resource graph*. A resource graph is a directed graph given by a four-tuple $\mathcal{G}_R = \langle \mathcal{V}_R, \mathcal{E}_R, f_R, g_R \rangle$ where $\mathcal{V}_R$ denotes the set of nodes or hosts and $\mathcal{E}_R \subseteq \mathcal{V}_R \times \mathcal{V}_R$ denotes the set of links connecting the hosts. The links can, but need not, represent physical connections. In general, they represent logical connections or paths over an internetwork which could be as simple as a single hop on a LAN.

$f_R : \mathcal{V}_R \to \mathbb{R}^s$, $s \geq 1$, is a function that characterizes the resource configuration on the hosts. For example, if $s = 2$, then the first component might refer to the number of processors or CPUs on a host and the second component may represent the clock rate. $g_R : \mathcal{E}_R \to \mathbb{R}^r$, $r \geq 1$, characterizes the property of a link which may include components such as bandwidth and link latency. f_R, g_R may be generalized to represent more complex resource characteristics (e.g., processors on a given host may possess different clock rates).

Let us define the *process relation graph* or simply *process graph* which depicts the various dependencies that processes possess both with respect to inter-process couplings and other software dependencies. A process graph is a directed graph given by a four-tuple $\mathcal{G}_P = \langle \mathcal{V}_P \cup \mathcal{V}'_P, \mathcal{E}_P, \gamma, \lambda \rangle$ where $\mathcal{V}_P$ denotes the set of processes, $\mathcal{V}'_P$ denotes the set of all other software objects (e.g., files), and $\mathcal{E}_P \subseteq \mathcal{V}_P \times (\mathcal{V}_P \cup \mathcal{V}'_P)$ denotes the set of inter-process and process-to-software dependencies.

$\gamma : \mathcal{V}_P \to \mathbb{R}_+$ gives the *communication/computation ratio* of a process which is one way to capture its dynamic behavior. Thus for $k \in \mathcal{V}_P$, the larger $\gamma(k)$ the more I/O intensive process k is. $\lambda : \mathcal{E}_P \to \mathbb{R}_+$ is a refinement of γ and represents the coupling of processes to other processes or software resources. λ and γ satisfy

$$\gamma(k) = \sum_{e \in \mathcal{E}_P} \lambda(e), \qquad k \in \mathcal{V}_P,\ e \text{ contains } k.$$

Thus the process graph may be defined as a triple where γ is then defined in terms of λ.

4.2 Progress Rate and Optimal Scheduling

A *resource assignment* or simply *configuration* is a function $\xi : \mathcal{V}_P \cup \mathcal{V}'_P \to \mathcal{V}_R$ that assigns every process and software resource to a host, in general, many-to-one. In this paper, we will be concerned with the situation where all software resources belonging to $\mathcal{V}'_P$ are assigned to a fixed host. That is, for all $k \in \mathcal{V}'_P$ there exists $i \in \mathcal{V}_R$ such that for all ξ we have $\xi(k) = i$. Fixing such an assignment of $\mathcal{V}'_P$, let Ξ denote the set of all configurations ξ that obey the fixed assignment.

To introduce performance, we next define the *progress rate* of a process. The progress rate $V_\xi : \mathcal{V}_P \to \mathbb{R}_+$, given a configuration $\xi \in \Xi$, is a function that quantifies how fast a process $k \in \mathcal{V}_P$ progresses with its computation. For example, $V_\xi(k)$ may represent the number of process k's instructions executed per unit time given the resource assignment ξ. Clearly, progress rate will depend on configuration, and conversely, the "goodness" of a configuration may be measured by the progress rate that it induces over all processes in the system. This leads to the average progress rate, given configuration ξ, of all processes in the system

$$\bar{V_\xi} = \frac{\sum_{k \in \mathcal{V}_P} V_\xi(k)}{|\mathcal{V}_P|}.$$

For typical process and resource graphs and resource assignments $\xi \in \Xi$, we would expect the following monotonicity properties to be satisfied:

$$\frac{\partial V_\xi(k)}{\partial f_R(i)} \geq 0 \qquad \text{and} \qquad \frac{\partial V_\xi(k)}{\partial g_R(i)} \geq 0 \tag{4.1}$$

where $k \in \mathcal{V}_P$ and $f_R(i)$, $g_R(i)$ are the resource properties[6]. That is, as resources are increased—other things being equal—the progress rate of an application is not adversely affected if not helped. For some resource properties (e.g., link latency), the monotonicity relation may occur in the negative direction, e.g., increased link latency is "bad." However, by a change of variables, they can always be formulated in a uniform way. For some pathological cases, an increase in resources may result in a decrease in progress rate for one or more application processes. We will exclude such cases from the model.

[6]In (4.1) we have assumed, for simplicity of exposition, that $f_R(i)$, $g_R(i)$ are scalar quantities. Similarly, we use continuous notation to keep the cluttering of notation to a minimum. Their discrete counterparts are straightforward to derive.

The aforementioned development allows us to formulate a static optimization problem corresponding to the optimal resource allocation problem:

$$\max_{\xi \in \Xi} \bar{V}_\xi. \tag{4.2}$$

This problem can be shown to be NP-hard. The proof is omitted for brevity.

Theorem 4.3 *The decision problem corresponding to the optimization problem given in* (4.2) *is NP-hard.*

Thus far the resource allocation problem we have formulated is a static optimization problem. The dynamic optimization problem where both the process graph and resource graph are functions of time—e.g., load arrival process for process graph and structural perturbations for resource graph—is straightforward to formulate but even more difficult to attack than the static counterpart which is already NP-hard.

We take the quasi-stationary approach to solving dynamic optimization problems [34] where we assume that the nonstationary system can be decomposed, in time, into stationary segments during which a solution procedure to the stationary problem—incorporating both computation and communication costs—is applied.

4.3 Iterative Load Balancer

An elementary operation in the integrated scheduling algorithm will be the approximation of the progress rate difference of two configurations $\xi, \xi' \in \Xi$

$$d(\xi, \xi') = \bar{V}_\xi - \bar{V}_{\xi'}.$$

In particular, given a configuration ξ, we will be interested in the set of configurations

$$\mathcal{N}(\xi) = \{\, \xi' \in \Xi : \exists \ell \in \mathcal{V}_P, \forall k \in \mathcal{V}_P \setminus \{\ell\}, \xi(k) = \xi'(k) \text{ and } \xi(\ell) \neq \xi'(\ell) \,\}.$$

That is, $\mathcal{N}(\xi)$ consists of all configurations where the process-to-host assignment differs with ξ on a *single* process. This is, for example, the case when a single process is migrated from one host to another host for a given configuration.

Since $\bar{V}_\xi$, by definition, incorporates both computation and communication requirements as well as their costs under the configuration ξ, if $d(\xi', \xi) > \theta$ for some $\theta > 0$, then process migration as dictated by ξ, ξ' would be warranted to improve performance. This also implies that θ need only be a function of process migration cost—an easily estimable quantity—and the lifetime duration of processes which has been investigated in [40, 41]. For certain parallel computing applications, process lifetimes can be on the order of minutes, hours, if not days, and the process migration cost becomes a negligible factor.

4.4 Conjoint Computation/Communication Scheduling with Progress Rate

As a first step to estimating $d(\xi', \xi)$, let us consider the case of mutually independent or isolated processes that do not engage in interprocess communication with each other. Let $k, \ell \in \mathcal{V}_P$ be two such processes. By definition, $\lambda(k, \ell) = \lambda(\ell, k) = 0$. That is, their coupling is zero.

Fix two hosts $i, j \in \mathcal{V}_R$ where $\xi(a) \neq i, j$, for all $a \in \mathcal{V}_P$. That is, they are empty. Assume that all hosts engage in *head-of-line processor sharing* (PS) [36, 49] when scheduling processes, the basic template on which most OS processor scheduling algorithms are based. Assume a slotted system with equal time slots; this is used in modeling the ON/OFF (i.e., computation vs. blocking) behavior of processes where, during a time slot, a process is either ON or OFF.

Let ρ_i, $0 \leq \rho_i \leq 1$, be the utilization of host i and let $\rho_i(k)$ be the utilization attributable to process k. Clearly, if k is the *only* process running on i, then $\rho_i = \rho_i(k)$. We will denote this single-process-alone-on-a-host utilization by α_k (fix a reference host or assume all hosts are homogeneous). In all of the above, we assume "long-lived" processes in the sense of infinite lifetime and for which the ratio α_k exists. Also, note that

$$\alpha_k = \rho_i(k) = \lim_{T \to \infty} \frac{V_\xi(k)\, T}{L_T}$$

where T is the unit of time over which progress rate is measured, L_T is the maximum amount of work (e.g., number of instructions) executable within time T, and ξ is any configuration that assigns process k, and only k, to host i. When T is fixed, we will sometimes call $V_\xi(k)\, T / L_T$ the progress rate of process k when there is no confusion.

The basic question we wish to answer is: given two mutually independent processes $k, \ell \in \mathcal{V}_P$ with ratios α_k, α_ℓ, what are their progress rates $\rho_i(k), \rho_i(\ell)$ when jointly scheduled on the same host i. This is needed in estimating the predicted progress rate stemming from migrating process k onto the same host as process ℓ.

Lemma 4.4 (Two-Process Utilization) *Let $k, \ell \in \mathcal{V}_P$ be two mutually independent processes, i.e., $\lambda(k, \ell) = \lambda(\ell, k) = 0$, with single-process-alone-on-a-host ratios α_k, α_ℓ. Then if k, ℓ are jointly scheduled on host $i \in \mathcal{V}_R$ implementing PS processor scheduling,*

$$\rho_i(k) = \frac{\alpha_k}{1 + \alpha_k \alpha_\ell}, \qquad \rho_i(\ell) = \frac{\alpha_\ell}{1 + \alpha_k \alpha_\ell}, \tag{4.5}$$

with probability 1 where the probability space is with respect to picking (uniformly) randomly from the set of all processes with ratios α_k, α_ℓ, respectively.

Clearly, $\rho_i = \rho_i(k) + \rho_i(\ell) = (\alpha_k + \alpha_\ell)/(1 + \alpha_k \alpha_\ell)$. The lemma can be generalized to $n \geq 0$ processes. Since the α_k's are observable quantities, the progress rate of candidate configurations can be computed. Figure 4.1 illustrates the scheduling of two processes $k, \ell \in \mathcal{V}_P$ with their given single-process-on-a-host profiles and the resulting joint schedule under PS scheduling.

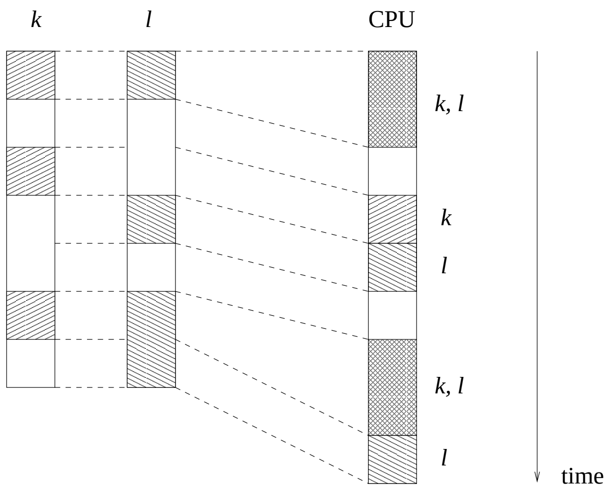


Figure 4.1: Joint scheduling of two processes k, ℓ with given single-process-alone-on-a-host profiles using PS and the resulting schedule.

Proof of Lemma 4.4. The trace of process k, when scheduled alone on host i, can be viewed as a random sequence

$$x^k = x_1^k x_2^k \cdots x_n^k \cdots$$

where $x_1^k, x_2^k, \ldots$ are i.i.d. binary random variables with $\Pr\{\, x_n^k = 1 \,\} = \alpha_k$, $\Pr\{\, x_n^k = 0 \,\} = 1 - \alpha_k$ where "$x_n^k = 1$" represents the event that at time slot n process k occupies the CPU and "$x_n^k = 0$" means that the CPU is idle. Similarly for process ℓ.

First, consider conjointly scheduling x_1^k and x_1^ℓ. Using PS scheduling, if the values are 00, then neither process requires the CPU and thus the first slot remains idle. If the values are 01 or 10, then only one process needs to access the CPU, the slot is allocated to one of the processes, and thus, is utilized. If the values are 11, PS scheduling dictates that processes k, ℓ be serviced each with rate $1/2$ during the next two time slots.

Let $y = y_1 y_2 \cdots y_m \cdots$ denote the random variable obtained by applying PS to x^k and x^ℓ iteratively. Based on the above, the process of obtaining y may be viewed as depending on an infinite trial of a four-sided coin toss with outcomes 00, 01, 10, 11, respectively, each with probability $(1-\alpha_k)(1-\alpha_\ell)$, $(1-\alpha_k)\alpha_\ell$, $\alpha_k(1-\alpha_\ell)$, and $\alpha_k\alpha_\ell$. Given $m \geq 0$, let $A_k(y, m)$ denote the total time interval—which must equal an integer number of time slots—in the prefix $y^m = y_1 y_2 \cdots y_m$ where process k has been scheduled. Then

$$\rho_i(k) = \lim_{m \to \infty} \frac{A_k(y^m, m)}{m}. \tag{4.6}$$

Assume n trials of the four-sided coin toss have been executed. Let B_{10}^n denote the number of 10 outcomes and let B_{11}^n denote the number of 11 outcomes. Then it follows that

$$m = n + B_{11}^n$$

where m is the length index from equation (4.6). On the other hand,

$$A_k(y^m, m) = B_{10}^n + B_{11}^n.$$

Thus, (4.6) can be rewritten as

$$\rho_i(k) = \lim_{n\to\infty} \frac{B_{10}^n + B_{11}^n}{n + B_{11}^n} = \lim_{n\to\infty} \frac{B_{10}^n/n + B_{11}^n/n}{1 + B_{11}^n/n}.$$

Since the four-sided coin toss sequence defines a multinomial distribution, B_{10}^n, B_{11}^n obey binomial distributions with $B_{10}^n \sim B(n, \alpha_k(1-\alpha_\ell))$, $B_{11}^n \sim B(n, \alpha_k\alpha_\ell)$. But, then, by the Strong Law of Large Numbers,

$$\Pr\{\lim_{n\to\infty} B_{11}^n/n = \alpha_k\alpha_\ell\} = 1,$$

and similarly for B_{10}^n. Hence,

$$\rho_i(k) \xrightarrow{\text{as}} \frac{\alpha_k(1-\alpha_\ell) + \alpha_k\alpha_\ell}{1 + \alpha_k\alpha_\ell} = \frac{\alpha_k}{1 + \alpha_k\alpha_\ell},$$

and by a symmetric argument $\rho_i(\ell) \xrightarrow{\text{as}} \alpha_\ell/(1 + \alpha_k\alpha_\ell)$. ■

The lemma—as far as we know—is not covered by previous results in PS scheduling due to the difference in arrival process modeling. Our model of a process—and quantified by α_k—is of a "process" where the ON periods are characterized as *instructions* be they from arithmetic calculation or from instructions needed to carry out I/O where CPU scheduling is partially involved. The OFF periods are *blocking* periods where a process is waiting on one or more events to occur, either related to communication or other interrupts. Thus this empty or idle period is *consequent* or *causal* to the last instruction executing and is not "forgetful."

In traditional task modeling [36, 49], the task arrival process is not of a single task in the blocking sense of above but rather of sequential tasks whose interarrival time obeys a fixed distribution (e.g., exponential, heavy-tailed). This implies that if two tasks arrive sequentially separated by some time interval T apart, then if the server or CPU is still busy with the first task after T time units have elapsed, the second task can be *immediately* scheduled after the first task has been serviced. This is not the case, however, for the *blocking task* model. Following is the generalization of Lemma 4.4 to n processes. The proof is a straightforward extension of the counting argument used in Lemma 4.4 and is omitted here for brevity.

Lemma 4.7 (n-Process Utilization) *Consider n mutually independent processes with single-process-alone-on-a-host ratios α_k, $k \in [1, n]$. Then if the n processes are jointly scheduled on host $i \in \mathcal{V}_R$ implementing PS processor scheduling,*

$$\rho_i(k) = \alpha_k\Big(1 + \sum_{h=2}^{n}(h-1)\sum_{s=1}^{\binom{n}{h}}\prod_{r=1}^{n}\alpha_r^{x_r(s)}(1-\alpha_r)^{1-x_r(s)}\Big)^{-1}, \qquad k \in [1, n], \tag{4.8}$$

with probability 1 *where* $x_r(s) \in \{0,1\}$ *and the binary vector* $(x_1(s), x_2(s), \ldots, x_n(s))$, $1 \le s \le \binom{n}{h}$, *respresents the s'th element in a canonical ordering of* $\binom{n}{h}$.

Clearly, when the number of processes n is "large" the expression given by (4.8) is computationally expensive to evaluate. However, in most instances, non-application processes (e.g., system processes and various management daemons), collectively, take up only a small fraction of CPU utilization unless thrashing or other degenerate situations arise. Thus the evaluation, in practice, need only be done for small values of n which makes computing $\rho_i(k)$ on-line feasible.

As an application of Lemma 4.7, consider migrating a process k from host i to host j $(i \neq j)$ where the former has n_i resident application processes and the latter has n_j. After migration host j will have $n_j + 1$ processes and i will have $n_i - 1$ processes. Since the single-process-alone-on-a-host ratios can be measured on both hosts, it is possible to use Lemma 4.7 to compute the predicted or estimated progress rate of the candidate configuration prior to actual migration, and if it is deemed larger than the current configuration's progress rate, issue a migration. Generally, given a configuration ξ, we estimate the progress rates associated with configurations belonging to $\mathcal{N}(\xi)$ and choose the maximal one if its predicted progress rate exceeds that of ξ.

4.5 Communication Dilation

The previous scheduling results apply to a group of mutually independent, i.e., $\lambda(k,\ell) = \lambda(\ell,k) = 0$, processes. That is, although each process is still blocking, the blocking does not stem from dependence with other processes in the group of processes considered. If there is coupling, however, splitting apart coupled processes can lead to the amplification of a process' blocking component thus reducing its single-process-alone-on-a-host ratio. Conversely, if coupled processes located at a distance are brought closer together—either in terms of physical proximity or effective bandwidth—then the process' blocking component contracts thus leading to an increase in the single-process-alone-on-a-host ratio. The dilation effect is illustrated in Figure 4.2.

Thus, for coupled processes, α_k, $k \in \mathcal{V}_P$, is a function of configuration ξ, i.e., $\alpha_k = \alpha_k(\xi)$. When moving from configuration ξ to configuration $\xi' \in \mathcal{N}(\xi)$, the resources assigned by ξ' to meet the couplings of migrated process k will change. For some aspects of process k the allocated resources—e.g., bandwidth or processor share—will have increased whereas for some other aspects they will have decreased. By relation (4.1), the reassignment of resources will affect k's progress rate. When using Lemma 4.7 to predict progress rate $V_{\xi'}(k)$ under the candidate configuration ξ', we need to first estimate $\alpha_k(\xi')$ to obtain an accurate estimate of $V_{\xi'}(k)$. Let

$$\alpha_k(\xi') = \delta\,\alpha_k(\xi).$$

We call $\delta \ge 0$ the *dilation factor* associated with process k with respect to the configuration pair (ξ, ξ'). The smaller δ, the more communication penalty is incurred by configuration ξ' vis-à-vis configuration ξ, and vice versa[7]. We estimate δ in the following way.

[7]To call δ "dilation factor" is, to some extent, a misnomer since if $\delta > 1$ then it affects a contraction.

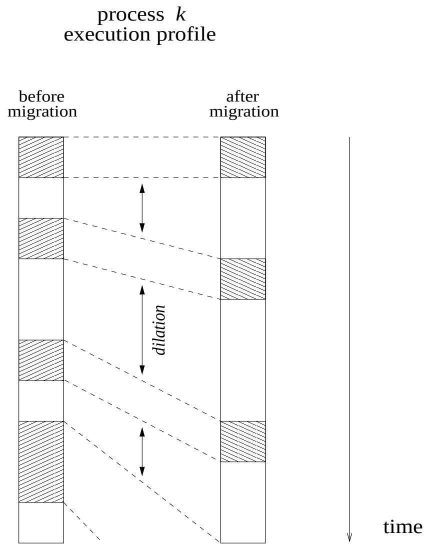


Figure 4.2: Effect of communication dilation on process k's execution profile due to migration to a "distant" host.

Let $\Lambda_{k,\ell}$ be the traffic rate—measured in bits per second (bps)—between process k and process ℓ. Let Λ_k be the total traffic rate impinging on k. $\Lambda_{k,\ell}$ is maintained dynamically by DUNES' run-time monitoring facility. Let

$$C_{i,j} \;=\; \sum_{\substack{a,b:\,\xi(a)=i,\\ \xi(b)=j}} \Lambda_{a,b}$$

where $i = \xi(k)$ and $j = \xi(\ell)$. That is, $C_{i,j}$ is the total traffic rate between the two hosts i, j where k and ℓ reside. Let $B_{i,j}$ be the total effective bandwidth between i and j. We compute δ by

$$\delta \;=\; \sum_{\substack{\ell=k \text{ or}\\ m=k}} \frac{\Lambda_{\ell,m}}{\Lambda_k}\,\delta_{\ell,m}$$

where $\delta_{\ell,m}$ is the dilation factor due to coupling between processes ℓ and m where one of them is k. That is, δ is computed as the weighted sum of the $\delta_{\ell,m}$'s where the latter are normalized by their corresponding traffic intensity. $\delta_{k,\ell}$, in turn, is estimated by

$$\delta_{k,\ell} = \frac{\Gamma(B_{i,j}/C_{i,j})}{\Gamma(B_{i',j}/C_{i',j})} \tag{4.9}$$

where $\Gamma(x) = x/(1-x)$ is the steady-state M/M/1 queue length formula, $i' = \xi'(k)$ is the new location of process k (note that since $\xi' \in \mathcal{N}(\xi)$, all other processes remain in their previous locations), and $C_{i',j}$, $B_{i',j}$ are the corresponding total traffic rate and effective bandwidth values for configuration ξ'.

Thus, $\delta_{k,\ell} = 1$ if configuration ξ' does not lead to a change in resources allocated to facilitating the coupling between k and ℓ. On the other hand, if the resources allocated to the coupling between

k and ℓ are shrunk—i.e., $B_{i',j}/C_{i',j} > B_{i,j}/C_{i,j}$, and hence, $\Gamma(B_{i',j}/C_{i',j}) > \Gamma(B_{i,j}/C_{i,j})$—then $\delta_{k,\ell} < 1$ and the consequences of increased communication cost are suffered accordingly. In fact, Γ amplifies this difference penalty the closer $B_{i,j}/C_{i,j}$ is to 1, i.e., network utilization is reaching saturation.

5 Performance Measurements

5.1 Experimental Set-Up

The experiments described in the following sections were conducted on dedicated LAN clusters in the Network Systems Lab (NSL) which is equipped with ten x86-based machines, each with a Pentium II processor at 399 MHz running SunOS 5.6, and four UltraSparc 1+ workstations running SunOS 5.5.1. These machines are connected via two 100 Mbps FastEthernet switches—one connecting the ten x86 machines and the other connecting the Sparc workstations—as well as a FORE ATM switch to which all UltraSparcs and two dual processor Intel machines are connected to. Some experiments requiring more machines were conducted in a separate lab equipped with twenty x86 machines, each with a Pentium processor at 90 MHz, running SunOS 5.5.1. These machines are connected via a 10 Mbps Ethernet.

```
main()
{
	st = getTimer();
	retval = system_call(arg1, arg2, ...);
	ed = getTimer();
	printf("Cost = %g\n", ed -  st);
}


double getTimer()
{
	struct timeval tp;
	gettimeofday(&tp, NULL);
	return (tp.tv_sec*1e6 + tp.tv_usec);
}
```

Figure 5.1: Sample code used to measure the overhead associated with the encapsulation of system calls. `system_call` can be `read`, `write`, `fork` and any number of other system calls.

All times reported in this section are wall clock times measured using the `gettimeofday` system call with microsecond granularity (see Figure 5.1). Using the wall clock time ensured that the overhead introduced by DUNES was taken into account. For some test cases such as when measuring the performance of `read` and `write` system calls where the cost depends on transient effects—e.g., availability of data in the local cache managed by DUNES—the performance cost measurements were amortized by repeating the operation a number of times (by default 100).

5.2 Cost of Process Migration

The cost of process migration consists of three parts, one, checkpointing a process image on the resident host, two, sending the checkpointed image and text over the network to the destination host, and three, restarting the process on the remote host. The overall migration cost, C_p, can be modeled as a linear equation

$$C_p = a\,S + b \tag{5.1}$$

where S is size of the checkpointed image (in Mbytes). The slope constant a, in turn, consists of three parts, $a = a_1 + a_2 + a_3$, where the first two components represent the checkpointing and restarting costs and the third component represents the network transfer cost. Of the three, the network transfer cost is the dominating cost and a_3 is estimated by tracking the available bandwidth between the source and destination hosts.

Figure 5.2 shows an instance of process migrating cost as a function of process image size when transferred over a 100 Mbps FastEthernet LAN. The estimated process migration cost—stationary during the migration experiments—is given by the linear equation $C_p = 0.49\,S + 0.59$. When using paging with copy-on-reference the initial transfer cost is significantly reduced, however, the overall cost accrued during the lifetime of a process as it changes its working set is incremental and strongly application dependent. The proper accredation of the latter is a more difficult task and under current investigation.

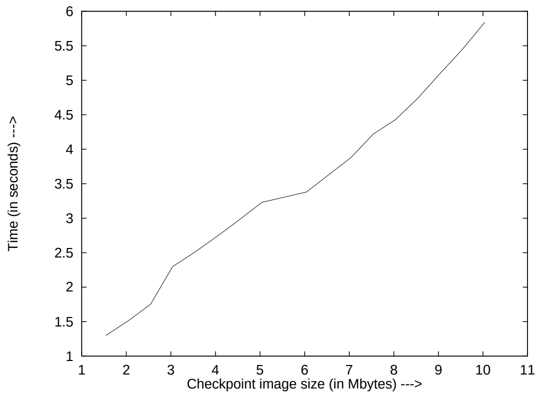


Figure 5.2: Cost of process migration as the image size of a process increases; $C_p = 0.49\,S + 0.59$.

5.3 Overhead Associated with Encapsulation of System Calls

In the following sections on performance measurements, we first identify the pure overhead incurred by DUNES as an additional software layer. We then proceed with measuring the effect of the various performance enhancement mechanisms, culminating in the demonstration of the overall net benefit of DUNES when the performance mechanisms are brought together by the communication-sensitive load balancer to achieve parallel application speed-up and increased system throughput.

As system calls in DUNES are encapsulated by our modified library, there are additional costs involved when system calls are invoked. Table 1 shows the overhead due to this encapsulation. The caching mentioned in the table corresponds to passive end-point caching. When caching is enabled, processes initiated on the home base have to contact the cache consistency manager when performing file access, thereby increasing the access times. In all other cases, the overhead observed are due to the interposed wrapper code to `syscall`. We note that this interposed code, which allows system calls to implement DUNES' functional and performance features, almost doubles the cost of system calls.

Method	open	lseek	read	write	fork (parent)	fork (child)
raw UNIX	86.0	27.7	22.6	24.2	1362.2	5406.8
caching disabled	129.0	47.7	42.0	46.0	5594.2	9057.7
caching enabled	959.8	641.5	618.3	622.9	5638.5	9135.0

Table 1: System call execution time (in microseconds) for processes on home base (no migration) in single host configuration.

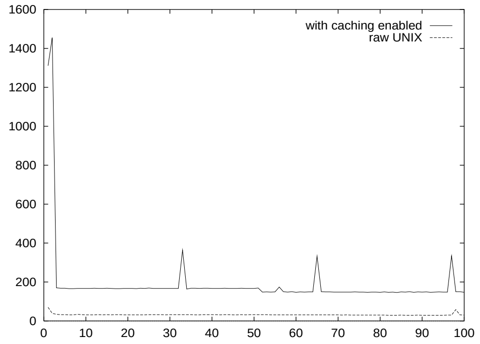
As a rule of thumb, system calls—even without the added overhead of DUNES—are known to be expensive and their frequent use is discouraged. Most application programs make infrequent calls to system calls, directly, or indirectly through the standard I/O library which adds further user-level processing before making one or more system calls. Moreover, the expensive nature of system calls comes from processing carried out in kernel mode such as when executing `read` or `write` with variable length payloads.

5.4 Cost of System Calls for Migrated Processes

Section 5.3 showed DUNES' overhead in its worst possible light, namely, when no parallelism is present and DUNES runs as a single host operating system. In this and following sections, we show the effect of DUNES' functional and performance features when two or more hosts are present and DUNES performs as a distributed operating system.

First, in a two host situation, for migrated processes, if caching is disabled, all system calls are routed to the home proxy through the remote proxy. On the other hand, if caching is enabled, the data is obtained from the remote proxy, assuming data is locally available. Cache misses can cause system calls to block and consequently slow down a process.

Passive end-point caching–disabled Table 2 compares the cost of executing system calls with and without passive end-point caching. When caching is disabled, each system call invocation incurs an overhead of sending messages to the home proxy to fetch data. In Table 2, the second row reflects this overhead.



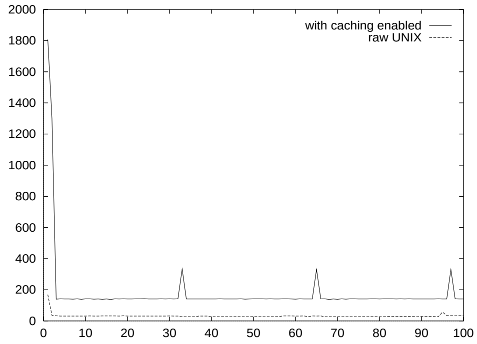


Figure 5.3: Cost of `read` (left) and `write` (right) system calls with caching enabled showing the effect of cache misses and flushes.

Passive end-point caching–enabled In Table 2, the third and fourth columns of the third row show that caching reduces the cost of read and write operations by a factor of 3. If the underlying network is slow or congested, the benefit of caching is further amplified. For system calls such as `open`, the home proxy needs to be contacted to keep the system in a consistent state. This increase in cost can be seen in the same table. When file offsets are shared by processes, each file access incurs an additional overhead of updating the offset maintained at the cache consistency manager. This effect can be discerned for the `lseek` system call. The cost of `fork` for a migrated process is about 6 times as high as a non-migrated process as it involves three separate `fork`s (cf. Section 3.2.1) and their initialization.

Method	open	lseek	read	write	fork (parent)	fork (child)
raw UNIX	86.0	27.7	22.6	24.2	1362.2	5406.8
caching disabled	1242.3	172.3	674.8	652.4	84689.3	32637.0
caching enabled	2053.7	5145.3	227.1	215.8	71468.0	29916.2

Table 2: System call execution time (in microseconds) for migrated processes in two host configuration.

Figure 5.3 shows the cost of each `read` and `write` system call over a total of 100 invocations. The amount of data read/written at each invocation was 32 bytes with a cache page size of 1024 bytes. For the `read` system call, the spikes in the plot correspond to cache misses and for the `write` system call, the spikes correspond to cache flushes. The spike at the beginning is unusually high due to the fact that the cache is initially empty.

Active end-point caching Table 3 summarizes the performance results for active end-point caching. Active end-points are cached using a push-based scheme. When data is available, it is pushed over to the host where the application process resides and then cached locally. Only `read` system calls benefit from active caching as writes have to be flushed immediately.

Figure 5.4: Completion time with increasing number of hosts to demonstrate the effects of load sharing. Left: Client/server communication set-up. Right: Completion time for active caching enabled and disabled.

On our testbed, without the presence DUNES, a `read` system call takes 8.4 μs and a `write` system call takes 7.6 μs to execute for a payload of 32 bytes. The row with *lcl write/rmt read* in Table 3 shows the effect of caching. Without caching, the cost of a `read` system call is 992.8 μs and with caching, it reduces to 173.8 μs which is a speed-up of 6. All other values are the same for both cache enabled and disabled cases.

5.5 Effect of Load Sharing

We increase the number of hosts participating in a concurrent application and show how this can affect performance measured by application completion time. Figure 5.4 (Left) shows the experimental set-up. We have one server process (S) and sixteen client processes (C) who communicate using `fifos`. The server process sends a series of messages to each client who, after some computation, send their results back to the server. This is a generic template for master/slave applications such as those arising in molecular sequence analysis and other application domains.

Initially, the server and clients run on a single host. Subsequent experiments involving multiple hosts migrate processes to other hosts when balancing load. Figure 5.4 (Right) shows completion

Caching of active end-points enabled

Method	read	write
lcl write/lcl read	24.8	21.6
rmt write/lcl read	1069.0	1073.8
lcl write/rmt read	173.8	23.7
rmt write/rmt read	1508.7	1509.2

Caching of active end-points disabled

Method	read	write
lcl write/lcl read	22.0	22.8
rmt write/lcl read	1070.7	1073.8
lcl write/rmt read	992.8	23.1
rmt write/rmt read	1530.3	1531.1

Table 3: `read` and `write` system call execution times (in microseconds) for active end-points. *lcl* refers to a process running on the home base (without migration) and *rmt* refers to a process running on a remote machine after process migration.

time as the number of participating hosts is increased from 1 to 16. We observe that due to communication overhead parallel speed-up saturates. The effect of active caching is discerned by the downward shift in the completion time curve. At the point of saturation (8 workstations), the performance gain due to active caching is about 50%.

5.6 Communication-Sensitive Load Balancing

5.6.1 Effect of Communication

Consider two application processes that repeatedly follow each computation phase with a communication phase to exchange their results. Instead of running the two processes on a single host, we can split the processes onto separate hosts thereby potentially decreasing completion time. (Note that the application processes still use a `fifo` to communicate even though they are on different machines due to DUNES' functional transparency mechanism.) The benefit of parallelism is affected by the amount of communication these processes engage in—the greater the level of communication, the lesser the potential benefit.

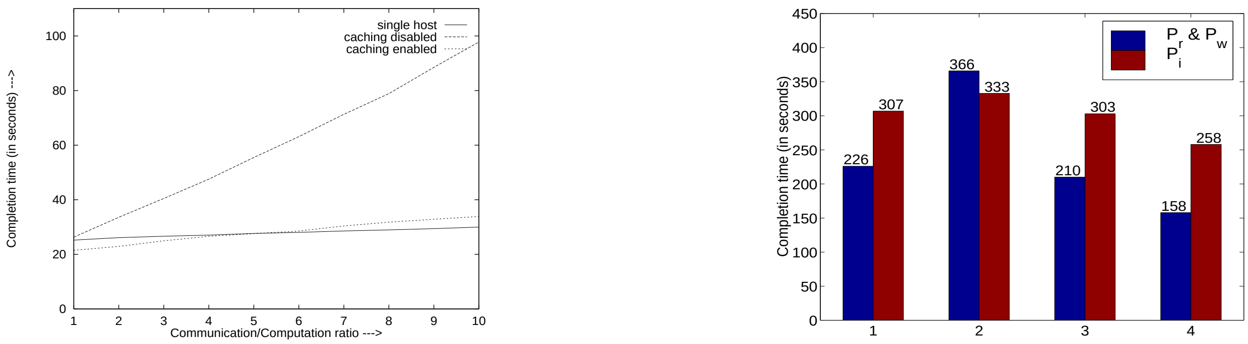


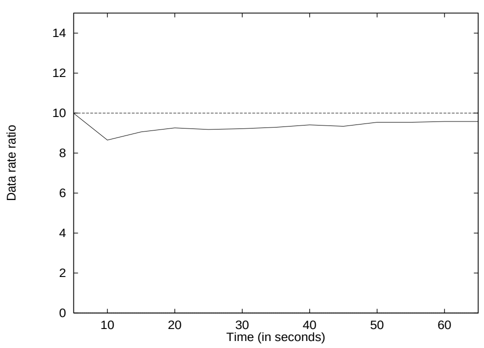
Figure 5.5: Left: Completion time with varying computation/communication ratio. Right: Three process scenario: P_w writes to P_r while P_i does no communication. Completion times when (1) no process is migrated, (2) P_w is migrated, (3) P_r is migrated, (4) P_i is migrated.

In Figure 5.5 (Left), we show the results of running two processes whose communication/computation ratio can be varied. We notice that as the relative amount of communication increases, in the single host scenario, the completion time also increases. In the two host scenario with one migrated process and active caching disabled, we observe that the communication cost—as the communication/computation ratio increases—is significantly amplified when caching is disabled. On the other hand, when active caching is enabled, for a communication/computation ratio up to 5, the completion time for the two host-two process set-up is smaller than that of the single host set-up. That is, the latency introduced by network communication is effectively hidden and a net parallel speed-up is achieved. When the communication/computation ratio is increased further, then it becomes detrimental to split the tightly coupled processes apart and scheduling on a single host becomes the optimal choice.

Figure 5.5 (Right) shows the results for a scenario where we have three processes, two of which, P_r and P_w, communicate with each other (P_w writes to P_r) while the third process (P_i) runs in isolation. If the coupling between P_r and P_w is nonnegligible, process P_i—other things being equal—is the best choice for migration. Figure 5.5 (Right) shows the completion times of the three processes for four configurations, one, when no process is migrated, two, when P_w is migrated, three, when P_r is migrated, and four, when P_i is migrated. We observe that migrating P_w yields the worst performance, giving completion times that are higher than when all three processes are scheduled on a single host. Migrating P_r gives a slight speed-up over the single host scenario, and migrating P_i yields superior performance via-à-vis all other cases.

5.6.2 Monitoring Communication

As each system call that performs I/O—on a per descriptor basis—has a counter, we can easily monitor the communication rate between processes. To demonstrate this, consider three processes where one process talks to the other two in a 1:10 ratio. The processes were run and the load balancer sampled the communication pattern every 5 seconds. The observed values are shown in Figure 5.6 (Left). We see that the measured data rate ratio is about 1:9.5, which is very close to the real ratio.



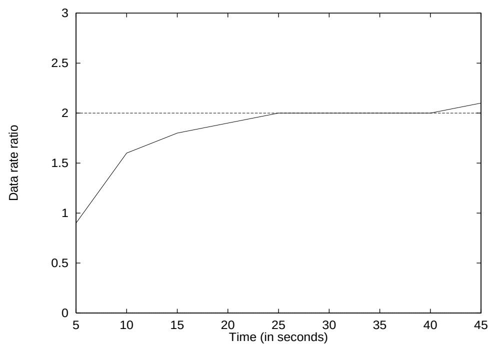


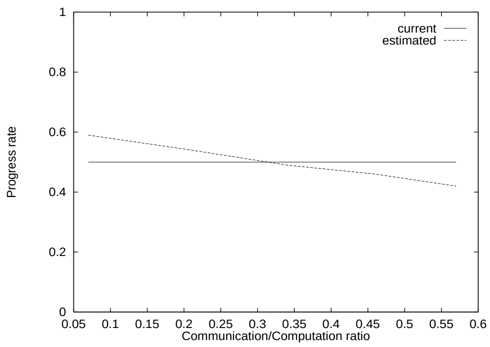
Figure 5.6: Measured data rate ratio for two process benchmark set-up for active (left) and passive (right) end-points.

To show the monitoring measurements for passive end-points, we considered two processes that accessed a file at different rates. One process accessed the file two times more frequently than the other process. Figure 5.6 (Right) shows that the monitored rate is close to the real rate as dictated by the application's intrinsic structure. The DUNES load balancer uses this and other run-time monitored information when making communication-sensitive load balancing decisions.

5.6.3 Load Balancing

DUNES' communication-sensitive load balancer dynamically balances load based on the progress rate measure. The load balancer estimates the progress rate associated with various candidate configurations and takes action only if the predicted progress rate of a candidate configuration is deemed larger than the measured progress rate of the current configuration.

To show how the communication/computation ratio can affect predicted progress rate, the resulting load balancing decisions, and ultimately application performance as measured by completion time, we consider a set-up involving two processes that communicate with each other, scheduled on a single host. We vary the relative frequency of writes and reads to get different communication/computation ratios as before.



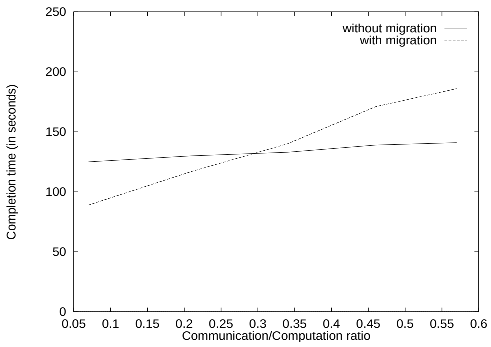


Figure 5.7: Left: For active end-points, the current measured progress rate and the predicted progress after migration are shown as a function of the communication/computation ratio. Right: The corresponding completion time for single host configuration vs. migrated configuration.

Figure 5.7 (Left) shows the measured progress rate of the current configuration—i.e., the two processes are scheduled on a single host—vs. the predicted or estimated progress rate when one of the processes is scheduled on a separate host as a function of the communication/computation ratio. The average per process progress rate for the single host configuration stays at 0.5 independent of the communication/computation ratio due to the fact that the processor on the single host is fully (and equally) utilized by the two processes, for both computation and I/O induced consumption of CPU cycles.

For the predicted progress rate for the migrated configuration, however, the estimated progress rate decreases as the communication/computation ratio increases reflecting the increased communication cost and its detrimental effect on splitting apart tightly coupled processes. Figure 5.7 (Right) shows the corresponding completion times as a function of the communication/computation ratio. Given an accurate estimate of the predicted progress rate, the completion times show that fruitful migrations can be initiated and unfruitful ones avoided.

5.7 Performance of Parallel Iterative Linear Equation Solver

5.7.1 Problem Domain

In this section, we show the performance of DUNES at facilitating parallel distributed computing for a parallel iterative procedure for solving linear equations. Consider the problem of solving a system of linear equations $Ax + b = 0$ where $A = (a_{ij})$ is an $m \times m$ matrix. Finding a numerical solution for x can be formulated as a fixed point problem [10] which, in turn, can be solved by the iterative procedure

$$x_i = -\frac{1}{a_{ii}} \left(b_i + \sum_{j=1}^{i-1} a_{ij} x_j + \sum_{j=i+1}^{m} a_{ij} x_j \right) .$$

If the spectral radius of A is less than 1, the iteration can be shown to converge.

A generic sample code used for implementing the iterative procedure is shown in Figure 5.8. After initialization, there is a loop within which a computation phase is followed by a communication phase followed by a `barrier` call to synchronize then followed by a termination check. Given n processes, each process is assigned m/n variables which it is responsible for updating. The updated values are then mutually exchanged using regular IPC (e.g., `fifo`). The `barrier` function call contacts a barrier server process and returns when the server process sends back a go-ahead message after synchronization.

```
main() {
    readInput();
    for(;;) {
        updateX();
        sendUpdates();
        receiveUpdates();
        barrier_sync();
        diff = computeDiff();
        if (terminate(diff))
            break;
    }
}
```

Figure 5.8: Sample code used in the parallel iterative algorithm to solve system of linear equations.

The generic code template shown above is indicative of many other applications amenable to parallel iterative solutions and thus is representative of a variety of computational procedures. The application programmer is oblivious to the workstation network computing environment wherein the application processes will be executed, writing the program with a single system image in mind. The only responsibility is one of indicating granularity (i.e., n). Arduous details of performing IPC and other standardized tasks can be alleviated with the help of user-level library routines that let the programmer focus on the computation part which is the core distinguishing component across different applications.

5.7.2 Parallel Application Speed-Up

Figure 5.9 (Left) shows application performance measured by average completion time as a function of granularity—i.e., number of processes—when all processes are scheduled on a single host vs. when each process is scheduled on a separate host by DUNES via process migration. The matrix in the benchmark problem instance tested was of size (3000×3000). The top plot for the single host schedule shows an increase in completion time as the number of processes is increased which is due to the overhead caused by IPC between processes on a single host. The bottom plot shows the completion times when DUNES is allowed to schedule processes by migrating each process to a separate host.

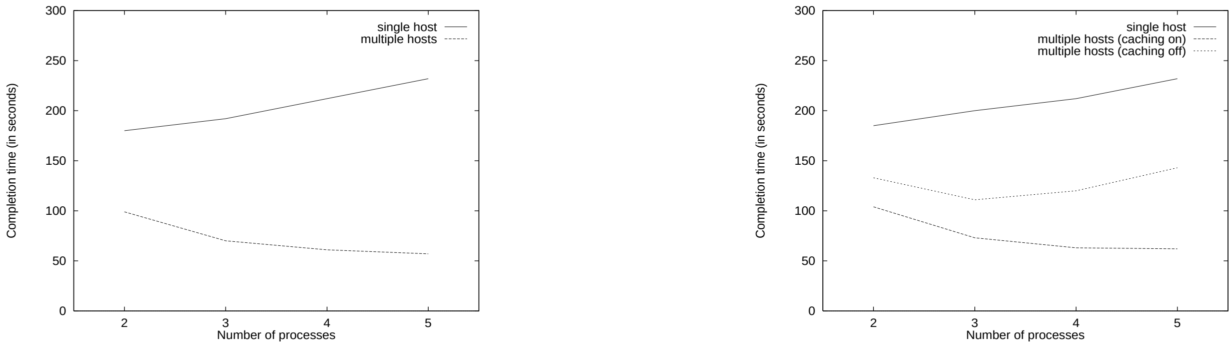


Figure 5.9: Left: Application completion time as a function of the number of processes participating in the computation when all processes are scheduled on a single host vs. when each process is scheduled on a separate host. Right: Application completion times for same set-up except that checkpointing of intermediate results is done periodically to achieve fault-tolerance.

Figure 5.9 (Right) shows application performance for the same set-up as before except that the application code of Figure 5.8 was augmented to implement periodic checkpointing of its intermediate results—e.g., to impart fault-tolerance when a computation is extremely long-lasting such as in cryptographic computations—which then induces periodic file I/O. Figure 5.9 (Right) shows that when all processes are scheduled on a single host, IPC overhead increases completion time, as before, and periodic file I/O causes the completion time curve to be shifted slightly upwards.

The middle plot shows completion times when each process is scheduled on a separate via migration, however, with passive end-point caching turned off. We observe that up to 3 processes (and hosts), the application experiences parallel speed-up. However, with four or more processes, the communication cost induced by writing the checkpointed intermediate values periodically to the home base begins to dominate and completion time increases henceforth. The bottom plot of Figure 5.9 (Right) shows application performance when DUNES' passive end-point caching mechanism is active. Client side passive end-point caching allows remote file I/O that would require network communication to be handled by local disk I/O and thus hide the communication latency.

5.7.3 Impact of Process Lifetime

Thus far we have only considered long-running processes, with or without dependencies, and the benefit gained by performing dynamic load balancing while incorporating the associated communication costs. In general, many types of processes are short-lived, and in such cases, the overhead due to migration can overshadow any gain obtained from a temporary increase of progress rate.

For example, in [40, 41] it is shown that the lifetime distribution of UNIX processes is heavy-tailed which carries the implication that most processes have very short life spans (e.g., utility commands such as ls executed under a shell). However, by the same token, heavy-tailedness also implies the existence of a few very long-lived processes who are then potential candidates for dynamic load balancing. In fact, heavy-tailedness implies the existence of nontrivial long-range correlation structure [57] which can then be exploited for prediction purposes. In essence, if we observe a process running for some time interval T that is "not too small," then the conditional probability that the process will continue to run for a "very long" while is high.

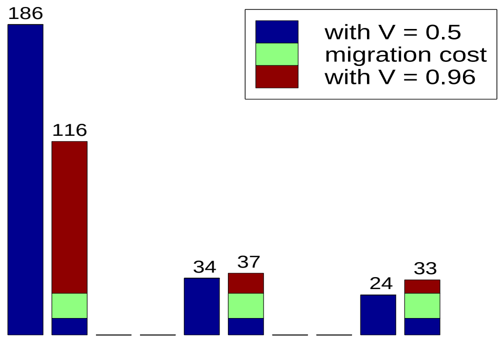


Figure 5.10: Completion time comparison of processes with varying life spans with, and without, migration.

Figure 5.10 shows completion time comparison of processes with varying life spans with, and without, migration. Our set-up has two processes participating in the iterative linear equation solver for a matrix of size (3000×3000) and process migration cost of 15 seconds. The process migration cost is only dependent on the image size, however, the lifetime of the iterative linear equation solver is also a function of the values in the matrix A, and hence by suitable varying the dominance of A—the smaller the spectral radius of A, the faster the convergence rate of the iterative solution procedure—the lifetime of the application can be controlled.

The left bar plot of Figure 5.10 shows the completion time for a particular instance of A when the two processes are executed on a single host (left bar) and when one of the processes is migrated to a separate host. The time when process migration is instantiated, the process migration cost, and the remaining completion time after migration are shown delineated (right bar). When the two application processes are scheduled on a single host, the average progress rate is 0.5 whereas if they are scheduled on separate hosts it jumps up to 0.96. The middle and right bar plots show instances of A which lead to successively shorter lifetimes, in fact, so much so that migrating a process—in spite of the increased progress rate of 0.96—leads to a higher completion time than leaving the processes on the same host. DUNES provides the means to estimate the progress of a candidate configuration and the associated progress migration cost, however, it does not provide a method for estimating the process lifetime which, in turn, is a function of progress rate. Process lifetime prediction is a subject matter onto its own of independent interest and beyond the scope of this paper.

5.7.4 Dynamics of Progress Rate Based Load Balancer

This section shows a distinguishing characteristic of dynamic load balancing based on the progress rate measure and its effect on performance.

First, consider 4 processes that work on solving a system of linear equations with two hosts available for processor sharing. Figure 5.11 (Left) shows the trace of the 4 processes being scheduled by DUNES' communication-sensitive load balancer. Initially, the 4 processes reside on a single host with the second host being idle. After 10 seconds, one of the processes in migrated to the idle host. The migration is triggered by a progress rate calculation that dictates that migrating a process to the idle host will increase overall progress rate. Subsequent to the first migration, another progress rate calculation for the candidate configuration involving migrating a second process reveals that this is beneficial and a further migration is triggered which leaves two processes on each host. As a result of this sequence of dynamic load balancing decisions, the processes terminate after 126 seconds. Suppose we did not intiate any migration. Then the completion time would have been 187 seconds. On the other hand, if we had stopped after one migration, then the completion time would have been 154 seconds.

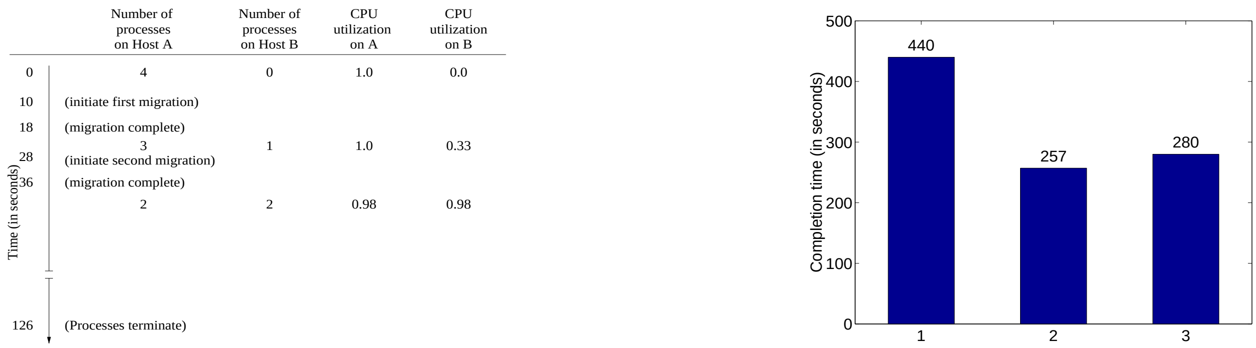


Figure 5.11: Left: Trace of 4 processes solving a system of linear equations with 1000 variables. Migration of processes is triggered by the estimated progress rate. Right: Completion time of 4 processes when (1) scheduled a single host, (2) processes are migrated one at a time to separate hosts, (3) two of the four processes are migrated to a separate host simultaneously.

The previous trace is not surprising as it agrees with the heuristic approach of "balancing" the load by putting two processes a piece on each host which, in this case, is the optimal load assignment. Suppose we have 4 processes running in isolation—i.e., they do not communicate with each other—on a single host. We can either let all four processes run on a single host, or migrate two of the four processes to the idle host simultaneously (corresponding to the above situation), or migrate one process at a time to the second host when the progress rate estimation dictates so. Other things being equal, one may expect migrating two processes simultaneously to the idle host to be the optimal load assignment. We show that this need not be the case.

Consider 4 processes each of which requires 1 time unit (it can be any fixed, equal time unit) to complete. When scheduling two processes on each host, the average completion time is 2 time units $((2+2+2+2)/4 = 2)$. On the other hand, if we migrate one process at a time—as dictated by the progress rate calculation—then the average completion time is 1.67 $((1 + 5/3 + 2 + 2)/4 = 1.67)$. Thus when each process is long-lived, the absolute performance difference between the two schedules can be significant. Figure 5.11 (Right) shows measurements of an isolated benchmark application (not the linear equation solver) and the resulting completion times when all processes are scheduled on a single host, when processes are scheduled according to the progress rate measure—i.e., $4/0 \mapsto 3/1 \mapsto 3/0 \mapsto 2/1 \mapsto 2/0 \mapsto 1/1 \mapsto 0/0$—and when two processes are simultaneously placed on the idle host. We observe that progress rate based scheduling yields the optimal assignment.

6 Conclusion and Discussion

We have described DUNES—a library distributed operating system—its features, both functional and as it pertains to performance. DUNES extends the functional capabilities of previous user-level dynamic load balancing systems, in particular, Condor, achieving transparent dependency maintenance in the presence of process-to-process and process-to-file couplings while preserving single processor UNIX semantics.

DUNES implements a number of performance enhancement features including push-based caching of active and passive end-points, demand paging of process checkpoints, and communication-sensitive load balancing all aimed at reducing the communication cost associated with migrated processes. Communication-sensitive load balancing implements a form of cost/benefit analysis based on a measure of "goodness" of resource allocation configurations called progress rate which, with the help of run-time monitored system state, allows detrimental load balancing actions to be avoided and potentially fruitful actions to be instantiated.

We have shown performance measurements of an implementation of DUNES for Solaris UNIX on LAN-based workstation networks which, starting with a quantification of its raw overhead, progressed toward showing its performance benefit with respect to application completion time and system throughput. The main thrust of future work is directed at extending the communication-sensitive load balancing model to incorporate real-time CPU scheduling to facilitate both guaranteed and best-effort services to time-constrained and QoS-sensitive applications.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[2] R. Agrawal and A. Ezzat. Processor sharing in NEST: A network of computer workstations. In *Proc. 1st IEEE Conference on Computer Workstations*, pages 198–208, 1985.

[3] P. Arbenz, M. Billeter, P. Guentert, and P. Luginbuehl. Molecular dynamics simulations on Cray clusters using the SCIDDLE-PVM environment. *Lecture Notes in Computer Science*, 1156, 1996.

[4] M. Ashraf Iqbal, J. H. Saltz, and S. H. Bokhari. A comparitive analysis of static and dynamic load balancing strategies. In *Proc. Int. Conf. on Parallel Processing*, pages 1040–1047, 1986.

[5] C. F. Baillie, G. Carr, L. Hart, and T. Henderson. Comparison of shared memory and distributed memory parallelization strategies for grid-based weather forecast models. In *Proceedings of the Scalable High-Performance Computing Conference, May 23–25, 1994, Knoxville, Tennessee*, pages 560–567, 1994.

[6] H. Bal, J. Steiner, and A. Tanenbaum. Programming languages for distributed computer systems. *ACM Computer Surveys*, 21(3):262–322, 1989.

[7] A. Barak and A. Shilow. A distributed load balancing algorithm for a multicomputer. *Software Practice and Experience*, 15(9):901–913, September 1985.

[8] Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX distributed operating system: load balancing for UNIX*, volume 672 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1993.

[9] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proc. 15th ACM Symp. on Operating System Principles*, pages 267–284, 1995.

[10] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, 1989.

[11] K. Birman and R. Cooper. The isis project: Real experience with a fault tolerant programming system. *ACM Operating Systems Review, SIGOPS*, 25(2):103–107, April 1991.

[12] K.P. Birman and T. Clark. Performance of the Isis distributed computing system. Technical Report TR-94-1432, Cornell Univ., Computer Science Dept., June 1994.

[13] Matt Bishop, Mark Valence, and Leonard F. Wisniewski. Process migration for heterogeneous distributed systems. Technical Report PCS-TR95-264, Dartmouth College, Computer Science, Hanover, NH, August 1995.

[14] R. K. Boel and J. H. van Schuppen. Distributed load balancing. In *Proc. 27th Conf. on Decision and Control*, page 1486, December 1988.

[15] Clemens Cap and Volker Strumpen. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, 19:1221–1234, 1993.

[16] N. Carriero and D. Gelernter. Application experience with linda. *Proceedings of the ACM/SIGPLAN PPEALS 1988*, 23(9):173–187, September 1988.

[17] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–459, April 1989.

[18] Jeremy Casas, Dan L. Clark, Ravi Konuru, Steve W. Otto, Robert M. Prouty, and Jonathan Walpole. MPVM: A migration transparent version of PVM. *Computing systems: the journal of the USENIX Association*, 8(2):171–216, Spring 1995.

[19] P. C. Chen. Climate and weather simulations and data visualization using a supercomputer, workstations, and microcomputers [2656-26]. In *Visual data exploration and analysis III: 31 January–2 February, 1996, San Jose, California*, pages 254–264, 1996.

[20] D. R. Cheriton. The V distributed system. In Akkihebbal L. Ananda and Balasubramaniam Srinivasan, editors, *Distributed Computing Systems: Concepts and Structures*, pages 165–184. IEEE Computer Society Press, Los Alamos, CA, 1992.

[21] Alex Cheung and Anthony Reeves. High performance computing on a cluster of workstations. In *Proc. First International Symp. on High-Performance Distributed Computing*, pages 152–160, 1992.

[22] S. Chowdhury. The greedy load sharing algorithm. *Journal of Parallel and Distributed Computing*, 9(1):93–99, May 1990.

[23] H. Clark and B. McMillin. Dawgs—a distributed compute server utilizing idle workstations. *Journal of Parallel and Distributed Computing*, 14(2):175–186, 1992.

[24] Partha Dasgupta, Richard LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. The Clouds distributed operations system. *IEEE Computer*, 24(11):34–44, November 1991.

[25] L. Dikken, F. van der Linden, J. J. J. Vesseur, and P. M. A. Sloot. DynamicPVM: Dynamic load balancing on parallel systems. In W. Gentzsch and U. Harms, editors, *High Performance Computing and Networking*, pages 273–277, Munich, Germany, April 1994. Springer Verlag, LNCS 797.

[26] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software Practice and Experience*, November 1989.

[27] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software–Practice & Experience*, 21(8):757–785, August 1991.

[28] W. Dzwinel and J. Blasiak. Pattern recognition via molecular dynamics on vector supercomputers and networked workstations. *Lecture Notes in Computer Science*, (919), 1995.

[29] D. Engler, M. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th ACM Symp. on Operating System Principles*, pages 251–266, 1995.

[30] M.R. Eskicioglu. Design issues of process migration facilities in distributed systems. *IEEE Comp. Soc. Techn. Comm. on Oper. Syst. and Appl. Env. Newsletter*, 4(2), 1990.

[31] Raphael Finkel and Yeshayahu Artsy. The process migration mechanism of Charlotte. *IEEE Computer Society Technical Committee on Operating Systems Newsletter*, 3(1):11–14, Winter 1989.

[32] B. Ford, K. Van Maren, J. Lepreau, S. Clawson, B. Robinson, and J. Turner. The FLUX OS toolkit: reusable components for OS implementation. In *Proc. 6th Workshop on Hot Topics in Operating Systems*, pages 14–19, 1997.

[33] U. Gaertel, W. Joppich, and A. Schueller. Medium-range weather forecast on parallel systems. In *Proceedings of the Scalable High-Performance Computing Conference, May 23–25, 1994, Knoxville, Tennessee*, pages 388–391, 1994.

[34] Robert G. Gallager. A minimum delay routing algorithm using distributed computation. *IEEE Trans. Commun.*, COM-25(1):73–85, 1977.

[35] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine.* The MIT Press, 1994.

[36] E. Gelenbe and I. Mitrani. *Analysis and synthesis of computer systems.* Academic Press, New York, 1980.

[37] D. Ghormley, D. Petrou, S. Rodrigues, A. Vahdat, and T. Anderson. GLUnix: a globale layer Unix for a network of workstations. To appear in *Software: Practice and Experience*, 1998.

[38] R. Guy, G. Popek, and T. Page. Consistency algorithms for optimistic replication. In *Proc. IEEE International Conference on Network Protocols*, 1993.

[39] A. Hac and Th.J. Johnson. Sensitivity study of the load balancing algorithm in a distributed system. *Journal of Parallel and Distributed Computing*, 10:85–89, 1990.

[40] M. Harchol-Balter. Process lifetimes are not exponential, more like $1/t$: implications on dynamic load balancing. Technical report, EECS, University of California, Berkeley, 1996. CSD-94-826.

[41] M. Harchol-Balter and A. Downey. Exploiting process lifetime distributions for dynamic load balancing. In *Proceedings of SIGMETRICS '96*, pages 13–24, 1996.

[42] A. Heddaya and K. Park. Mapping parallel iterative algorithms onto workstation networks. In *Proc. 3rd IEEE International Symposium on High-Performance Distributed Computing*, pages 211–218, 1994.

[43] A. Heddaya and K. Park. Parallel computing on high-speed wide-area networks: a pricing policy for its communication needs. In *Proc. 3rd IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, pages 188–191, 1995.

[44] A. Heddaya and K. Park. Congestion control for asynchronous parallel computing on workstation networks. *Parallel Computing*, 23:1855–1875, 1997.

[45] A. Heddaya, K. Park, and H. Sinha. Using Warp to control network contention in Mermera. In *Proc. 27th Hawaii International Conference on System Sciences, Maui, Hawaii*, pages 96–105, 1994.

[46] C. Jacqmot, E. Milgrom, W. Joossen, and Y. Berbers. Unix and load-balancing: A survey. In *Proc. EUUG '89*, pages 1–15, April 1989.

[47] M. Kaashoek, D. Engler, G. Ganger, H. Brice no, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proc. 16th ACM Symp. on Operating System Principles*, 1997.

[48] J. O. Kephart, T. Hogg, and B. A. Huberman. Dynamics of computational ecosystems. *Physical Review A*, 40(1):404–421, 1989.

[49] Leonard Kleinrock. *Queueing Systems, Volume 2: Computer Applications.* Wiley, New York, 1976.

[50] W. Leland and T. Ott. Load-balancing heuristics and process behavior. In *ACM Performance Evaluation Review: Proc. Performance '86 and ACM SIGMETRICS 1986, Vol. 14*, pages 54–69, May 1986.

[51] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proc. 8'th International Conference on Distributed Computing Systems*, pages 104–111, 1988.

[52] Jaroslav Mackerle. Implementing finite element methods on supercomputers, workstations and PCs: A bibliography (1985-1995). *Engineering Computations*, 13(1), 1996.

[53] K. I. Mandelberg and V. S. Sunderam. Process migration in UNIX networks. In *USENIX Technical Conference Proceedings*, pages 357–363, Dallas, TX, February 1988.

[54] M. N. Nelson, B. Welch, and J. Ousterhout. Caching in the sprite network file system. *Preprints for the Eleventh ACM Symposium on Operating Systems Principles*, pages 34–47, November 1987.

[55] John K. Ousterhout, A. R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *Computer*, 21(2):23–36, February 1988.

[56] M. Parashar, S. Hariri, A. Mohamed, and G. Fox. A requirement analysis for high performance distributed computing over LAN's. In *Proc. First International Symp. on High-Performance Distributed Computing*, pages 142–151, 1992.

[57] K. Park, G. Kim, and M. Crovella. On the relationship between file sizes, transport protocols, and self-similar network traffic. In *Proc. IEEE International Conference on Network Protocols*, pages 171–180, 1996.

[58] Kihong Park. Warp control: a dynamically stable congestion protocol and its analysis. *Journal of High Speed Networks*, 2(4):373–404, 1993.

[59] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. 15th ACM Symp. on Operating System Principles*, pages 79–95, 1995.

[60] Stefan Petri and Horst Langendörfer. Load balancing and fault tolerance in workstation clusters – migrating groups of communicating processes. *Operating Systems Review*, 29(4):25–36, October 1995.

[61] K.W. Ross and D.D. Yao. Optimal load balancing and scheduling in a distributed computer system. *Journal of the ACM*, 38(3):676–690, July 1991.

[62] Oliver Sharp. The grand challenges. *BYTE Magazine*, 20(2), February 1995.

[63] J.M. Smith. A survey of process migration mechanisms. *Operating Systems Review*, 22(3):28–40, July 1988.

[64] Jonathan M. Smith. A survey of process migration mechanisms. *ACM Operating Systems Review*, 22(3):28–40, July 1988.

[65] Peter Smith and Norman C. Hutchinson. Heterogeneous process migration: The tui system. Technical Report TR-96-04, University of British Columbia. Computer Science, February 1996.

[66] Volker Strumpen. Parallel molecular sequence analysis on workstations in the Internet. Technical Report 93.28, Department of Computer Science, University of Zurich, 1993.

[67] Michael Stumm. The design and implementation of a decentralized scheduling facility for a workstation cluster. In *Proceedings of the 2nd IEEE Conference on Computer Workstations*, pages 12–22, March 1988.

[68] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency, practice and experience*, 2(4):315–339, December 1990.

[69] Andrew S. Tanenbaum, Robert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experience with the Amoeba distributed operating system. *CACM*, 33(12):46–63, December 1990.

[70] L.G. Tesler. Networked computing in the 1990s. *Scientific American*, 265(3):54–61, September 1991.

[71] Amjad Umar. *Client/Server Internet Environments.* Prentice-Hall, 1997.

[72] R. Van Engelen and L. Wolters. A comparison of parallel programming paradigms and data distributions for a limited area numerical weather forecast routine. In *Conference proceedings of the 1995 International Conference on Supercomputing, Barcelona, Spain, July 3–7, 1995*, pages 357–364, 1995.

[73] L. Wolters, G. Cats, and N. Gustafsson. Limited area numerical weather forecasting on a massively parallel computer. In *Supercomputing '94: International conference — July 1994, Manchester*, pages 289–296, 1994.

[74] G. Yagawa, A. Yoshioka, S. Yoshimura, and N. Soneda. A parallel finite element method with a supercomputer network. *Computers and Structures*, 47(3), May 1993.

[75] Edward R. Zayas. Attacking the process migration bottleneck. In *Proceedings of the eleventh ACM Symposium on Operating System Principles*, pages 13–24, Austin, Texas, November 1987. ACM.

[76] S. Zhou and D. Ferrari. An experimental study of load balancing performance. In *Proc. IEEE Int. Conf. on Distr. Processing*, volume 7, pages 490–497, September 1987.