Towards Performance-Driven System Support for Distributed Computing in Clustered Environments[1]

John Cruz[2] and Kihong Park[3]

Network Systems Lab
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, U.S.A.

With the proliferation of workstation clusters connected by high-speed networks, providing efficient system support for concurrent applications engaging in nontrivial interaction has become an important problem. Two principal barriers to harnessing parallelism are: one, efficient mechanisms that achieve transparent dependency maintenance while preserving semantic correctness, and two, scheduling algorithms that match coupled processes to distributed resources while explicitly incorporating their communication costs. This paper describes a set of performance features, their properties, and implementation in a system support environment called DUNES that achieves transparent dependency maintenance—IPC, file access, memory access, process creation/termination, process relationships—under dynamic load balancing. The two principal performance features are push/pull-based active and passive end-point caching and communication-sensitive load balancing. Collectively, they mitigate the overhead introduced by the transparent dependency maintenance mechanisms. Communication-sensitive load balancing, in addition, affects the scheduling of distributed resources to application processes where both communication and computation costs are explicitly taken into account. DUNES' architecture endows commodity operating systems with distributed operating system functionality while achieving transparency with respect to their existing application base. DUNES also preserves semantic correctness with respect to single processor semantics. We show performance measurements of a UNIX based implementation on Sparc and x86 architectures over high-speed LAN environments. We show that significant performance gains in terms of system throughput and parallel application speed-up are achievable.

Key Words: Distributed operating systems, communication-sensitive load balancing, workstation networks, process migration, parallel distributed computing

1. INTRODUCTION

1.1. Motivation

With the advent of high-speed networks connecting a large number of high-performance workstations via high-speed local area and wide area networks, harnessing their collective power for distributed computing, including parallel computing, has become a viable goal. 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—efficient mechanisms for transparent dependency maintenance and resource scheduling which explicitly incorporate communication cost. Although these issues arise in parallel machines as well, their impact is amplified in workstation networks requiring new solutions.

With respect to transparent dependency maintenance, we seek to design efficient mechanisms that allow the flexibility to perform dynamic resource scheduling—in particular, load balancing—when it is deemed beneficial to do so without the mechanism itself becoming a burdensome cost factor. The mechanism should preserve semantic correctness when scheduling tasks across distributed resources and it should be easily deployable, to the extent possible, on commodity computing platforms.

With respect to communication-sensitive resource scheduling, the lack of special calibrated communication facilities in the form of interconnection networks renders a workstation cluster more susceptible to communication overhead including congestion effects. In the case of load balancing, balancing of processor load without proper regard for communication costs can deteriorate performance when network communication becomes a

[1] Supported in part by NSF grant ESS-9806741.

[2] Additionally supported by a fellowship from the Purdue Research Foundation (PRF). E-mail: cruz@cs.purdue.edu.

[3] Additionally supported by NSF grants ANI-9714707, ANI-9875789 (CAREER), and grants from PRF and Sprint. E-mail: park@cs.purdue.edu, tel.: (765) 494-7821.

dominant factor. A principal lesson learned from load balancing is that processes best suited for migration are those that are 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—single host interprocess communication or file access is turned into network communication—as well as the overhead associated with process migration itself.

To harness parallelism in workstation networks effectively, an integrated system support approach is needed that achieves transparent dependency maintenance efficiently and makes load balancing decisions based on explicit consideration of both computation and communication costs.

1.2. New Contributions

The contributions of this paper are twofold. First, we design a system support environment called DUNES (**D**istributed **UN**ix **E**xten**S**ion) which achieves transparent dependency maintenance of coupled processes under dynamic load balancing. DUNES is implemented as a user-level distributed operating system which exports a single system image to the user and achieves single processor UNIX semantics. Second, we design a set of performance enhancement features that mitigate the overhead introduced by the user-level transparent dependency maintenance mechanisms. Communication-sensitive load balancing facilitates 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 costs.

1.2.1. Transparent Dependency Maintenance

As part of the distributed OS functionality, our system implements dynamic process migration following the user-level mechanism employed in Condor [23]. However, unlike in Condor, our dynamic process migration mechanism handles *dependencies* arising from interprocess communication, file access, memory access, process creation, and process

relationships, maintaining transparent bindings consistent with single processor UNIX semantics.

The first form of transparency is *functional transparency* where DUNES ensures that various forms of dependencies including process-to-process and process-to-file dependencies 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 transparent to the process. The second 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.

1.2.2. Performance Features

Although implementing transparent dependency maintenance at the user-level that achieves single processor UNIX semantics is a nontrivial technical challenge, the resulting support mechanisms would be of limited utility if their associated overhead were significant. We mitigate the overhead cost by employing active and passive end-point caching as part of DUNES' mechanism design. At the algorithmic level, we use communication-sensitive load balancing to affect dynamic scheduling which explicitly accounts for communication costs arising from dependencies.

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 (or processes) without waiting for the issuance of reads. The coordination of multiple readers to a single `pipe` or `fifo` when engaging in destructive `read` is one of the issues arising in this context.

Passive end-point caching Similarly to active end-point caching, we seek to reduce 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 single processor UNIX semantics.

Communication-sensitive load balancing Whereas the aforementioned mechanisms try to reduce the cost of facilitating dependencies over a distance, communication-sensitive load balancing tries to, one, prevent strongly coupled processes—coupled with other processes, files, or memory—from being split apart in the first place if the benefit of parallelism is deemed less than its cost, and two, trigger migrations if they are deemed beneficial to do so. 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 initiate fruitful ones.

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 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 DUNES architecture including its functional and performance features. Section 4 shows performance results of a DUNES implementation for Solaris UNIX on both Sparc and x86 architectures measured over private, controlled high-speed LAN workstation networks. We conclude with a discussion of our results and future work.

2. RELATED WORK

A myriad of software support environments have been advanced with a view toward facilitating concurrent applications in workstation environments [2, 4, 6, 10, 15, 23]. 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 [1, 9, 13, 17, 20, 22, 28, 29, 31], key components 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 [5, 8, 18, 19, 25, 26]. 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 can lead 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 the gain obtained from a more balanced load.

Distributed operating systems, for the most part, are written from scratch and are kernel-based with a well-defined interface to kernel 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 *library OS* [21] approach to imparting resource management functionalities 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 are purposely implemented using user-level libraries. 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—portability from network computing and efficiency from distributed operating systems.

Our system is similar to Condor [23] 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, file access, memory access, and process creation while maintaining transparent bindings consistent with UNIX semantics. Condor is restricted to migrating "stand-alone" processes with support for remote file access using a network file system. The ability to facilitate transparent dependency maintenance is important from the perspective that present day applications tend to engage in some form of interaction—frequent or infrequent—and maintaining dependencies correctly with respect to single processor UNIX semantics is an important requirement.

Another related system is GLUnix [16]. GLUnix, however, does not support process migration and dynamic load balancing. As with Condor, it does not possess the performance enhancement features of DUNES. A similar observation holds for PVM [30], an execution environment for development and execution of large concurrent and parallel applications that consist of many interacting, but relatively independent, components. MPVM [7] and

DynamicPVM [12] are extensions to PVM that support process migration. They follow the approach used by Condor to checkpoint and restart processes.

3. STRUCTURE OF DUNES

DUNES (**D**istributed **UN**ix **E**xten**S**ion) is a distributed operating system designed using the approach of *library operating systems* [14]. Distributed OS functionality is injected into a commodity OS—our implementation is in the context of Solaris 2.6—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.

3.1. Functional Features

3.1.1. Transparent Processor Sharing

Process migration 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 [23, 24, 27] and we follow an analogous strategy in our own implementation.

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 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, respond to `timer_create`'s alarms for periodic logging of run-time monitored 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.

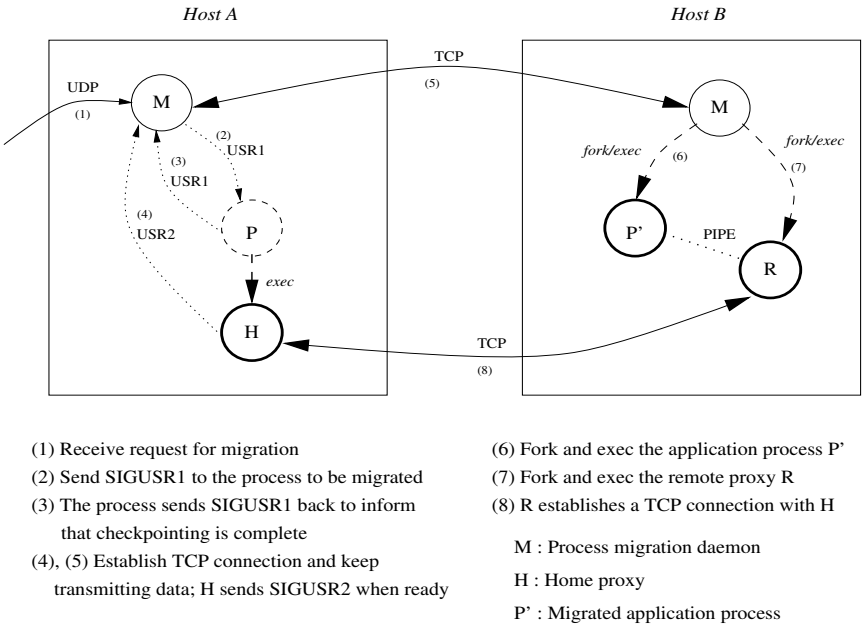


FIG. 3.1. Migration of process P from host A to host B and resulting triangle relation between migrated process P' via remote proxy R to home proxy H.

Basic system configuration 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 dependencies transparently. This is done by `exec`ing the proxy code from inside the application process—the last action of 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 on the home base, is able to transparently handle dependency relations.

If a migrated process is subsequently migrated again, then all state information on the previous remote host is deleted and the new configuration reached is isomorphic to 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 3.1.

3.1.2. Transparent Dependency Maintenance

Functional transparency 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), network communication, and shared memory access. Furthermore, a process, after migration, may fork off one or more processes which can complicate the dependency structure significantly.

Condor [23] supports transparent file access but does not allow process migration in the presence of IPC, network communication, or process creation. One practical justification for this is that, other things being equal, the processes that benefit most from migration are isolated processes. 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. DUNES provides the flexibility to engage in dynamic scheduling through the support of transparent dependency maintenance mechanisms, and the issue of whether for certain processes it would be beneficial to migrate is left to an *algorithmic* component—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, and refraining from doing so if it is deemed detrimental.

Semantic transparency Another important aspect of maintaining dependencies transparently is the issue of correctness. 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 exporting a single system image. 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.

Transparency mechanism 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 (cf. Figure 3.1). 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*. The latter executes the system 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 running. 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 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*.

3.1.3. Shared Memory Segments

DUNES handles accesses to *shared memory segments*—i.e., segments created and shared through `shmget` and `mmap` system calls—transparently after processes have migrated. Memory segments created through `mmap` which are marked *private* are no different from data segments. Due to this data consistency and integrity follow immediately. These segments are treated as ordinary data segments except that they are flushed back to disk once the mapping is removed.

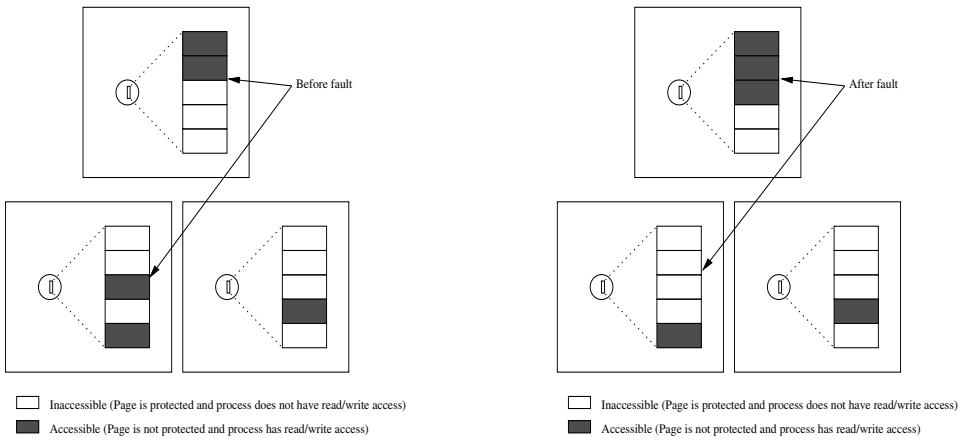


FIG. 3.2. Three processes on different machines sharing the same memory segment. Left: Status before an access fault. Right: Status after changing the access permissions and transferring the page

Memory segments that are created through `mmap` with the segments marked *shared* and `shmget` require special treatment as we have to maintain data consistency and provide access that is consistent with single processor UNIX semantics. Figure 3.2 illustrates the basic mechanism. A process has access to only those pages in its virtual address space that are marked accessible. Any access to a page that is marked inaccessible will result in a segmentation fault, which is caught by a signal handler. If the access is to a valid address, DUNES identifies the process that has possession of the page and a signal is sent to the page owner to relinquish the page: in essence, set the page as inaccessible. It is then transferred to the process that faulted. When control returns from the signal handler, the process continues with access to the page restarting at the instruction that caused the segmentation fault.

3.1.4. 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 process-to-process and process-to-file communication go through system calls, by implementing simple counting mechanisms inside `read`, `write`, 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 queried to obtain both long-term and short-term utilization information.

3.2. Performance Features

3.2.1. 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 (e.g., link latency and physical bandwidth) or logically (e.g., 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 may 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 manifesting as 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, 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, care must be taken to prevent one reader from being starved by another as well as making sure that the cached data is shipped to the correct destination given that for IPC and network communication `read` is destructive. Active end-point caching not only hides communication latency, it also enables scheduling actions involving process migration to be fruitful when, without, the same actions may be determined to be detrimental. This leads to further opportunities for performance improvement that would otherwise not be accessible.

Figure 3.3 shows a typical scenario involving a read-shared `fifo` accessed by two or more migrated processes. One of the migrated processes' proxies takes charge as the cache manager. The other proxies contact the cache manager to get the cached data. When applications running 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.

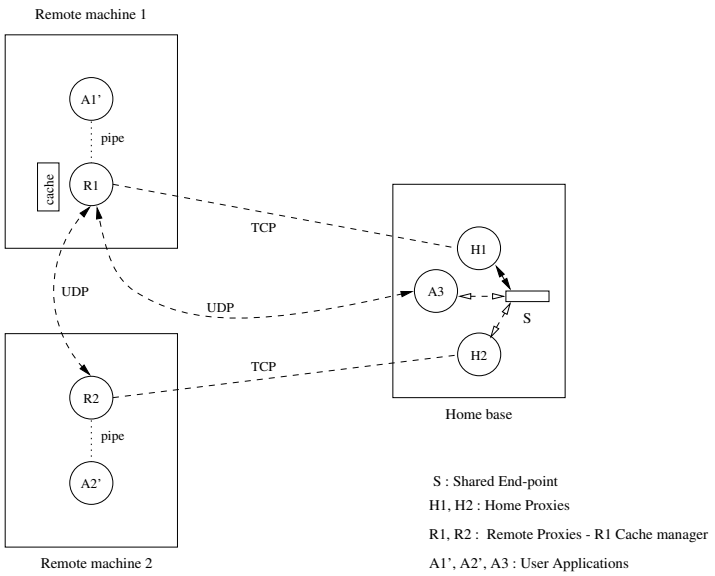


FIG. 3.3. Active end-point caching.

3.2.2. 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 migration. 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 for generality—optimizations that interface with particular network file systems, if present, are in development—but rather engages in its own push-based caching scheme. We use a system with k pages, page size S, and LRU page replacement policy. The overall structure is shown in Figure 3.4.

DUNES implements single writer/multiple reader semantics using a cache consistency manager which conforms to single processor UNIX semantics. The granularity of access is on a per-page basis, and as is generally the case, 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.

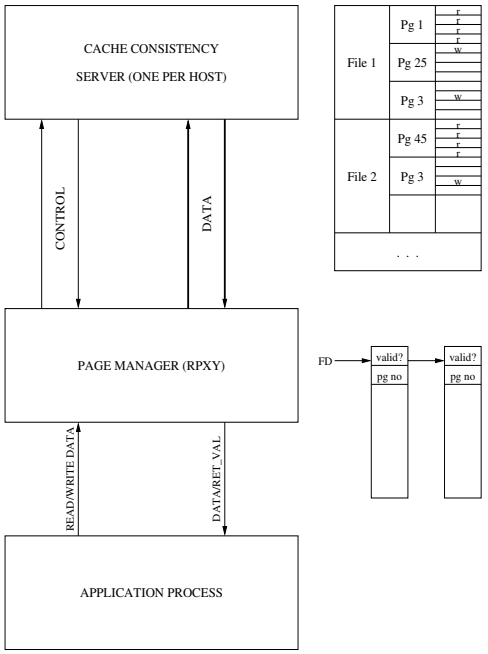


FIG. 3.4. Passive end-point caching. The proxy process on the destination machine RPXY acts as the page manager.

3.2.3. Communication-Sensitive Load Balancing

A principal lesson learned from dynamic load balancing is that processes best suited for migration are those that are 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 which may entail 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. A formal description of progress rate based dynamic load balancing and its properties can be found in [11]. We describe its relevant features in Section 4 along with performance results.

4. PERFORMANCE MEASUREMENTS

4.1. 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. 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.

All times reported in this section are wall clock times measured using the `gettimeofday` system call with microsecond granularity. 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).

4.2. Overhead of DUNES' Functional Mechanism

4.2.1. Overhead Associated with Encapsulation of System Calls

We identify the pure overhead incurred by DUNES as an additional software layer. As system calls in DUNES are encapsulated by our modified library, there are overhead costs involved when system calls are invoked. This is shown in Table 1. 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 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.

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

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

4.2.2. Cost of System Calls for Migrated Processes

Section 4.2.1 showed DUNES' overhead in its worst possible light, namely, when no parallelism is present and DUNES runs as a single host operating system. In the following, we show the performance effect of DUNES when two or more hosts are present and DUNES functions as a distributed operating system. First, in a two host situation, for

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

Passive end-point caching 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. The second row of Table 2 reflects this overhead. When caching is *enabled*, the third and fourth columns of the third row of Table 2 show a cost reduction 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 and their initialization.

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

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

Active end-point caching Figure 4.1 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 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.

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

FIG. 4.1. `read` and `write` system call execution times (in microseconds) for active end-points.

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 size of 32 bytes. The row with *lcl write/rmt read* in Table 4.1 shows the effect of caching. *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. 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 the enabled and disabled cases.

4.2.3. Active/Passive End-Point Caching and Parallel Speed-Up

We increase the number of hosts participating in a concurrent application and show how this can affect performance measured by application completion time. Figure 4.2 (Left) shows the experimental set-up. We have one server process (S) and 16 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 the clients run on a single host. Subsequent experiments involving multiple hosts migrate processes to other hosts when balancing load. Figure 4.2 (Right) shows completion 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%. Similar results hold for passive end-point caching.

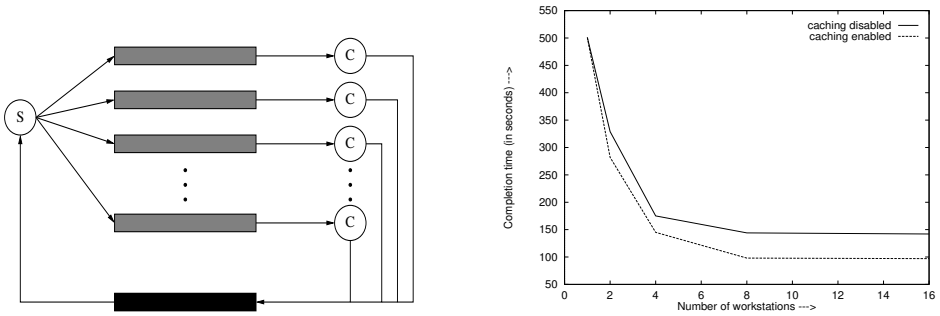


FIG. 4.2. Left: Client/server communication set-up. Right: Completion time for active caching enabled and disabled.

4.3. Run-Time Monitoring of 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. Consider three processes where one process talks to the other two in a 1:10 ratio. The communication pattern of the processes were sampled every 5 seconds at run-time. The measured values are shown in Figure 4.3 (Left). We see that the observed data rate ratio is about 1:9.5, which is close to the real ratio. To show the monitoring measurements for passive end-points, we consider two processes where one process accesses the file at twice the rate as the other process. Figure 4.3 (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 run-time monitored information when making communication-sensitive load balancing decisions.

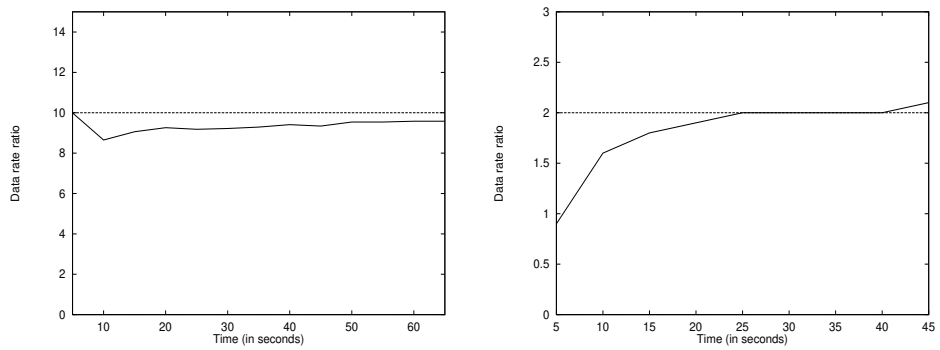
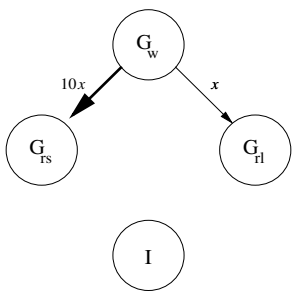


FIG. 4.3. Measured data rate ratio for two process benchmark set-up for active (left) and passive (right) end-points.

4.4. Communication-Sensitive Load Balancing

4.4.1. Set-Up

Figure 4.4 (Left) shows the set-up used in the next experiments. There are three collaborating processes—G_w, G_{rs} and G_{rl}—and one independent process (I) that are initially resident on the same host. G_w is a producer that performs some computation and writes the result to G_{rs} and G_{rl}, the consumers. The latter read the data written by G_w and perform some computation. The data rate among these processes, however, is not uniform. The data rate between G_w and G_{rs} is 10 times that of G_w and G_{rl}. Process I, on the other hand, does not communicate with either of the three processes. It has a CPU utilization of 0.65 when run in isolation.




```
GetLoad()
Src = getHostWithHighestLoad();
Dst = getHostWithLeastLoad();
If (highest_load - least_load < THETA1) {
      return;
}
victim = chooseProcessToMigrate();
If (estimated_new_diff(Src, Dst) > THETA2) {
      issueMigrateRequest(victim);
}
```

FIG. 4.4. Left: Process communication structure. G_w is a producer which writes to G_{rl} and G_{rs}. The amount of writes to G_{rs} is 10 times that of G_{rl}. I is a process running in isolation. Right: Template code used by the heuristic load balancing algorithms.

4.4.2. Impact of Communication Cost

Consider the case where we have two hosts with all four processes running on one host. Figure 4.5 (Left) compares the result of migrating a single process to the idle machine vs. the case when there is no migration. When process I is migrated, its completion time is reduced from 146 sec to 64 sec. The completion time of the group G_w, G_{rl}, G_{rs} reduces from 330 to 294 seconds. On the other hand, if we migrate G_{rs} (the strongly coupled reader), the completion time decreases only marginally. We observe that if we migrate G_{rl} instead of G_{rs}, the completion time decreases significantly. The aforementioned experiments show that, other things being equal, the weaker the coupling, the larger the performance gain obtained from migrating a process. Figure 4.5 (Right) shows performance results for the case when we increase the number of hosts by one and two processes are migrated instead of one. We observe that the best combination is $\{I, G_{rl}\}$ followed by $\{I, G_{rs}\}$ and $\{G_{rl}, G_{rs}\}$.

Process Migrated	Completion Time (sec)	
	G_w, G_{rl}, G_{rs}	I
None	330	146
I	294	64
G_{rl}	270	116
G_{rs}	321	146

Processes Migrated	Completion Time (sec)	
	G_w, G_{rl}, G_{rs}	I
None	330	146
$\{I, G_{rl}\}$	237	63
$\{I, G_{rs}\}$	289	64
$\{G_{rl}, G_{rs}\}$	277	117

FIG. 4.5. Left: Completion times in a two host scenario with only one process being migrated to the idle host. Right: Completion times in a three host scenario where two processes are migrated.

4.4.3. Heuristic Load Balancing

We implement three heuristic load balancing schemes that are successively more sensitive to communication costs. All three algorithms have the same structure as depicted in Figure 4.4 (Right). The algorithms collect load information from the participating machines

and identify the least utilized and most utilized machines as target and source machines. If the difference in CPU utilization on these two machines differs by an amount greater than a threshold θ_1, then we choose a process from the source machine for migration. Each heuristic has its own way of selecting a process for migration—`chooseProcessToMigrate()`—and that is where they differ. After a process is chosen for migration, all schemes perform a final check before initiating migration. This is accomplished by testing if the predicted difference in CPU utilization or *imbalance*—which is obtained by subracting the CPU utilization of the migrating process from the source machine and adding it to the target machine, then taking their resulting difference—is greater than the current imbalance by some threshold θ_2. If not, the migration is cancelled. The rationale behind this test is based on the progress rate load balancing algorithm where a more comprehensive cost/benefit analysis is performed prior to process migration. Following are descriptions of `chooseProcessToMigrate()` for the three algorithms:

Algorithm 1 This is the simplest scheme which does *not* incorporate communication cost information; i.e., it makes load balancing decisions based on CPU utilization only. In particular, it chooses a process which has maximum CPU utilization.

Algorithm 2 This algorithm makes use of both CPU utilization and communication cost where the latter is captured by the *coupling count* of a process—the number of file descriptors open in the process. *Algorithm 2* tries to select a process which is as loosely coupled or independent as possible. The key assumption is that, the greater the number of open descriptors, the tighter its coupling. The specific criterion is based on *average normalized rank* where, given a set of nonnegative numbers $a_1, a_2, \ldots, a_k$, the *normalized rank* γ_{a_i} of a_i, $i \in [1, k]$, is defined as

$$\gamma_{a_i} = \frac{a_i - a_*}{a^* - a_*}$$

with $a_* = \min_{j \in [1,k]} a_j$ and $a^* = \max_{j \in [1,k]} a_j$. Given two sets of numbers and a pair of numbers a_i, b_j belonging to their respective sets, the average normalized rank $\bar{\gamma}_{a_i,b_i}$ is

defined as

$$\bar{\gamma}_{a_i,b_i} = \omega\gamma_{a_i} + (1-\omega)\gamma_{b_i}$$

where $0 \leq \omega \leq 1$. We form the two sets by choosing the CPU utilization numbers and coupling count (reordered) of processes belonging to the source host with $\omega = 1/2$. We then compute the average normalized utilization for each process and choose a process with the maximum value.

Algorithm 3 *Algorithm 2* assumes that the number of open file descriptors is proportional to the amount of communication. In general, this need not be the case. A process could have many file descriptors open and may not use any of them. Similarly, a process could have just one connection open through which it performs frequent data transfer. To capture the quantitative data rate, *Algorithm 3* uses the run-time monitored per-descriptor data rate to compute the total data rate of processes. It then chooses the process with a maximal average normalized rank where the latter is computed using the total data rate in place of coupling count.

4.4.4. Performance Comparison of Algorithms 1, 2, and 3

We compare the three schemes in the same set-up described above with the four processes G_w, G_{rl}, G_{rs}, and I initiated on a single host.

Two host scenario We examine the actions of the three algorithms for a two host scenario where on one host the four processes are initiated and the other is idle. *Algorithm 1*—which only considers CPU utilization—selects G_w to migrate. G_w happens to be the most strongly coupled process. It is also the one with the highest CPU utilization which renders it least suited for migration due to the dominance of communication cost. *Algorithm 2* chooses I to migrate as it is the process that has a high CPU utilization but also has the least coupling count. After I terminates, *Algorithm 2* selects G_{rs} to migrate which is a less suitable choice than G_{rl} due to the latter's weaker coupling with respect to data rate. *Algorithm 3*, which uses the monitored data rate instead of the coupling count, makes the optimal

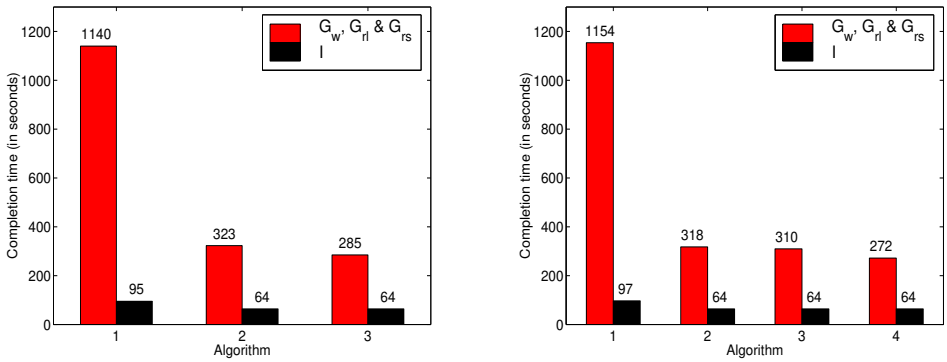


FIG. 4.6. Left: Completion times on two host scenario. Right: Completion times on three host scenario. The process structure is given in Figure 4.4 (Left).

decision by choosing I followed by G_{rl}. This also results in the smallest completion time. The performance results are shown in Figure 4.6 (Left).

Three host scenario Consider the set-up above, however, with two idle hosts instead of one. *Algorithm 1* selects G_w as before. After migration, the CPU utilization of G_{rl} and G_{rs} decrease significantly due to their dependence on G_w prompting *Algorithm 1* to migrate I next. *Algorithm 2* selects I and G_{rs} to migrate to the idle hosts—in that order—and after I terminates, migrates G_{rl}. *Algorithm 3* initially moves I and G_{rl} to the two idle hosts, and upon completion of I, migrates G_{rs}. Although *Algorithm 3* yields the best performance, the optimal decision—shown as *Algorithm 4* in Figure 4.6 (Right)—is not to migrate G_{rs} after I terminates. This shows the potential for further performance improvement by incorporation of refined communication and computation information.

4.5. Parallel Iterative Linear Equation Solver

4.5.1. Problem Domain

In this section, we show the performance of DUNES at facilitating parallel distributed computing in the context of 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 which 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 [3].

A generic sample code used for implementing the iterative procedure is shown in Figure 4.7. After initialization, there is a loop within which a computation phase is followed by a communication phase. At the end of the latter, a `barrier` call for synchronization followed by a termination check are executed. 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;
    }
}
```

FIG. 4.7. Sample code used in the parallel iterative algorithm to solve system of linear equations.

4.5.2. Parallel Application Speed-Up

Figure 4.8 (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.

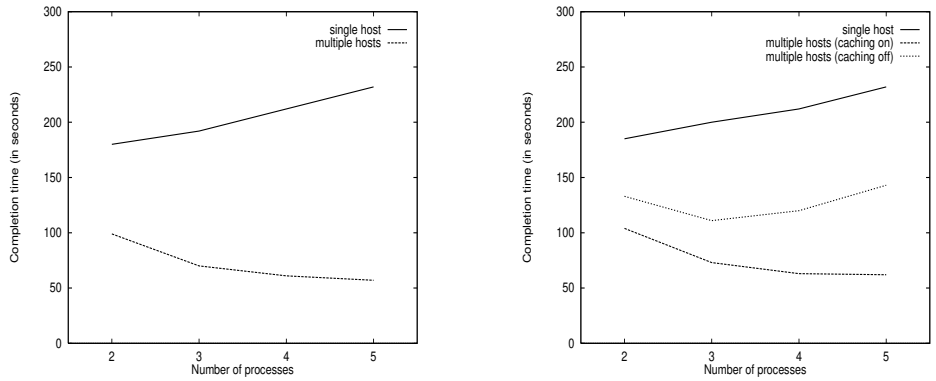


FIG. 4.8. Left: Application completion time as a function of the number of processes participating in the computation when all processes are scheduled on a single host vs. when each process is scheduled on a separate host. Right: Application completion times for same set-up except that checkpointing of intermediate results is performed periodically to achieve fault-tolerance.

Figure 4.8 (Right) shows application performance for the same set-up except that the application code of Figure 4.7 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. We observe that when all processes are scheduled on a single host, IPC overhead amplifies completion time and periodic file I/O causes the completion time curve to shift upwards. The middle plot shows completion times when each process is scheduled on a separate host 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 4.8 (Right) shows application performance when DUNES' passive end-point caching mechanism is active. Client-side passive end-point

caching allows file I/O that would require network communication to be handled by local disk I/O and thus hide the communication latency.

4.5.3. Dynamics of Progress Rate Based Load Balancer

We show the dynamics of load balancing based on the progress rate measure and its effect on performance. Consider four processes that work on solving a system of linear equations with two hosts available for processor sharing. Figure 4.9 shows the trace of the 4 processes being scheduled by DUNES' communication-sensitive load balancer based on progress rate. Initially, the four processes reside on a single host with the second host idle. After 10 seconds, one of the processes is migrated to the idle host. The migration is triggered by a progress rate calculation that dictates that migration will increase overall progress rate. Subsequent to the first migration, another progress rate calculation reveals that migrating a second process is beneficial which triggers a further migration. As a result of this sequence of dynamic load balancing decisions, the processes terminate after 126 seconds. Without any process migration, the completion time is 187 seconds. With a single process migrating, the completion time is 154 seconds.

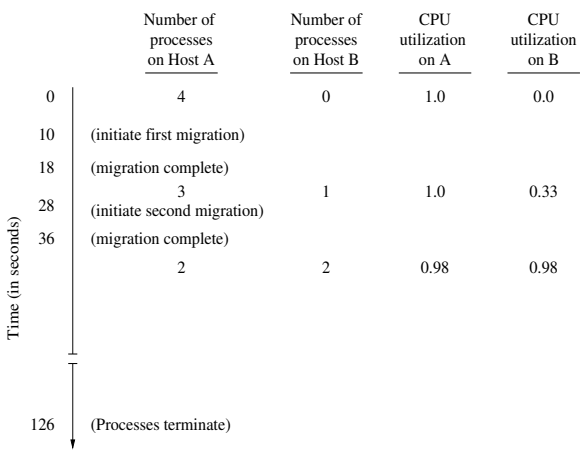


FIG. 4.9. Trace of 4 processes solving a system of linear equations with 1000 variables.

5. CONCLUSION

We have described DUNES, a library distributed operating system, organized arounds its two principal features—transparent dependency maintenance and performance enhancement features—aimed at mitigating the former's overhead and exploiting its facilitation of dynamic load balancing of coupled applications. We have shown that user-level system support for dynamic scheduling of coupled processes is feasible with active/passive endpoint caching reducing the communication cost associated with dependency maintenance, and communication-sensitive load balancing affecting improved performance with respect to application completion time and system throughput. The main thrust of current work is directed at extending the communication-sensitive load balancing model to incorporate real-time CPU scheduling—built on top of Solaris RT mode—to facilitate both guaranteed and best-effort services to time-constrained and QoS-sensitive applications.

REFERENCES

1. 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.

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

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

4. 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.

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

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

7. 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.

8. 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.

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

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

11. J. Cruz and K. Park. Towards performance-driven system support for distributed computing in clustered environments. Technical Report CSD-TR-98-035, Department of Computer Sciences, Purdue University, 1998.

12. 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.

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

14. 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.

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

16. 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.

17. 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.

18. 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.

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

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

21. 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.

22. 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.

23. 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.

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

25. 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.

26. Kihong Park. Warp control: a dynamically stable congestion protocol and its analysis. In *Proc. ACM SIGCOMM '93*, pages 137–147, 1993.

27. 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.

28. 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.

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

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

31. 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.