Transparent Migration and Rollback for Unmodified Applications in Workstation Clusters

Stefan Petri[1*] · Matthias Bolz[2] · Horst Langendörfer[2]

[1]Institute of Computer Engineering, Medical University at Lübeck
petri@iti.mu-luebeck.de

[2]Institute for Operating Systems and Computer Networks
Braunschweig University of Technology

Published as *Computer Science Report 98-02 of the Braunschweig University of Technology, Germany*

Abstract

Programmers and users of compute intensive scientific applications often do not want to (or even cannot) code load balancing and fault tolerance into their programs.

The PBEAM system [PL95, PSLS96] uses a global virtual name space to provide migration and rollback transparency in user space for distributed groups of processes on workstations. Applications always use the same virtual names for the operating system objects, independent of their current real location. The system calls are interposed and their parameters translated between the name spaces. Unlike other migration mechanisms, PBEAM does not require the applications to be written for a specific programming model or communication library.

The first approach to execute applications in the virtual name space was to link the programs with a modified system library. Now, in this paper we describe design and implementation of a separate system call interposition process [Bol97] that accesses the application via the debugging interface. The main advantage of this approach is that it can handle even unmodified (e. g. commercially bought) application programs. We compare measured performance figures with previous similar approaches [MS88, PSLS96] and the modified system library.

1 Motivation and Introduction

Networks of Workstations (NOWs) become increasingly attractive as platforms for parallel compute intensive applications, because their price/performance ratio is significantly better than that of massively parallel systems (MPPs) [KHG96, KLB98].

*At the time of writing funded by DFG contract SFB 342 at Institute for Computer Science, Munich University of Technology

However, in contrast to MPPs most NOWs operate in multi user mode, the nodes are shared between applications, and interactive users may not be disturbed by resource hungry background computations. These constraints make it desirable to use a dynamic load balancing facility that moves work from overloaded to idle nodes, or evacuates machines that are claimed for other purposes, e. g. interactive users. Additionally, the high probability of machine failures [LCP91, CCS95] necessitates fault tolerance measures for long running applications.

Programmers and users of compute intensive scientific applications are mainly interested in getting their problem solved. They expect load balancing and fault tolerance as services of the underlying operating or run time system and do not want to care about them in their application code. A solution for these problems is an application transparent migration and checkpointing system. Unfortunately, off-the-shelf workstation platforms provide these services only minimally, if at all.

In the next section, we give a brief overview of our PBEAM load balancing system for distributed applications on clusters of workstations. Because the basic problems of "freezing" the application's state and reviving it afterwards is the same in both cases, checkpointing or migration, PBEAM, like other systems (e. g. [LS92, Ste96]), handles them both. After that we focus on our approach to apply PBEAM transparently to unmodified binary application programs, and show some performance figures. We conclude after a short comparison to related work.
For a broader discussion of concepts and more technical details beyond this paper we refer to [PL95, PSLS96]. A shorter version version of this report has been published in the Proceedings of the 2^{nd} Workshop on Runtime Support for Parallel Processing at the 12^{th} International Parallel Processing Symposium (IPPS'98) [PBL98].

2 Overview of PBEAM

The goals of this project are to provide application transparent process migration and checkpointing / rollback for distributed applications on clusters of workstations, running in user space on unmodified Unix systems.

The state of a distributed computation consists of the states of its processes and the states of the communication links between them. We distinguish between a process' *internal* and *external* state [PL95]. The internal state consists of the address space and register contents, including program counter and stack pointer. The external state comprises the process' relation to the world outside its address space, the allocated resources, related processes and the communication peers. A process can manipulate its external state only through services of the system kernel. These services operate on objects like files, processes, communication endpoints, etc. They are invoked through system calls, whose arguments name the objects to operate on (file names, file descriptors, process numbers, transport addresses, etc.).

These names depend on the current location and time of the process execution. If a process is moved from one node to another, or an application is restarted from a checkpoint, the names for the objects will change. However, to provide transparency, the ap-

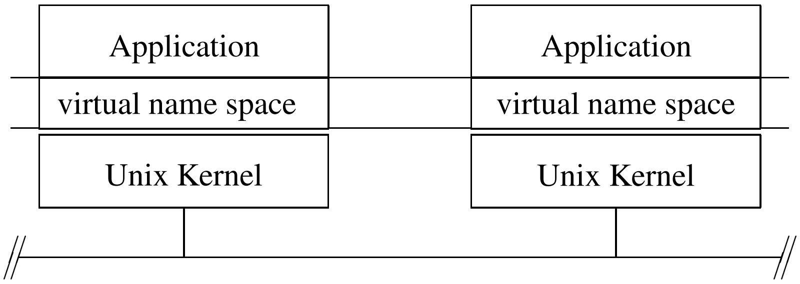

Figure 1 The virtual name space between kernel and application.

plication must be able to work with the same names regardless of being moved in space or time. Therefore, ⟨PBEAM introduces a system wide virtual name space for process IDs, transport addresses and file names. A virtual process table and virtual address tables are maintained for mapping between the virtual and current real names. The applications' system calls are interposed, the parameters are translated from the virtual into the underlying current real name space, and then the real system service is executed. In the same manner the return values are translated back from the real into the virtual name space. To avoid name collisions, the processes are assigned globally unique virtual process IDs. Through the system call interposition also the changes to the external state are tracked.

While interposing system calls with a modified system kernel promises the best transparency and performance, the dependency on a special operating system version is prohibitive to let a system be used widely [Jon92, PL95]. An early design decision was to not do any system kernel modifications [PL95, Jon92].

Basically, there are two possibilities to perform the system call interposition outside the kernel. Our first approach was a modified system call library. The system call functions are replaced by or wrapped into our own versions [CM92, LS92, Jon92]. It requires that the application can be linked with the modified library.

The other possibility is to control the application processes by aseparate controller process via the debugging interface. Most modern Unix flavors offer extensions to the "traditional" debugger interface that allow to stop a controlled process just before and after execution of a system call, i. e. when entering respectively exiting kernel mode [Sun90, FG91]. The main advantage of this approach is that it can work with applications that are available in binary form only, already completely linked, e. g. commercially bought programs.

In the ⟨PBEAM system we have separated the system call interposition component from the name space administration (the ⟨PBEAM demon "`pbeamd`" in fig. 2), thus making it easy to switch between different versions of both. To explore scalability, we have also implemented a centralized and a distributed version of the name space administration [PSB$^+$97]. Other components of the system are a scheduler and a dispatcher that executes the decisions made by the scheduler.

The system call interposition component has three main tasks, it gets its name from the first one: (i) During normal operation, it does the name space translations, as described above. (ii) When doing a checkpoint or migration, it has to capture the internal and external state, save it onto disk, via TCP to a checkpoint server, or to another machine

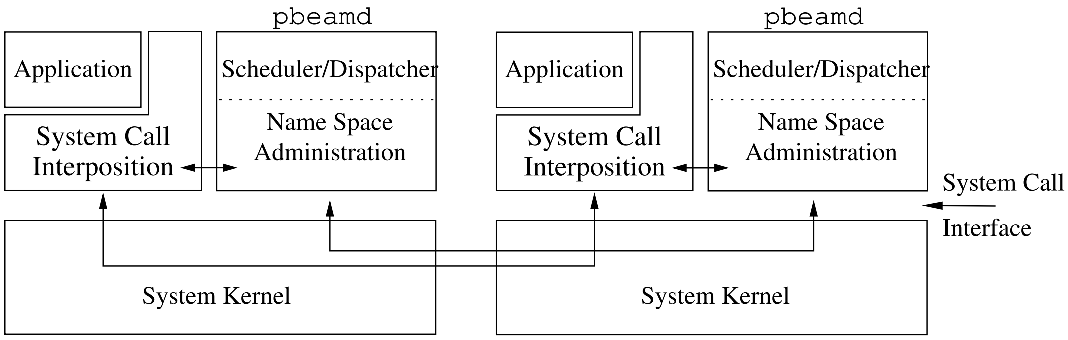

Figure 2 The architecture of PBEAM

for instant reconstruction (i. e. migration). (iii) Read a saved state from disk or from a network connection and reconstruct a running process from this state information.

3 Operation of the System Call Interposition Process (SCIP)

Our goal here is to provide the same rollback transparency and migratability as is achieved by linking an executable with the modified system library for applications that cannot be linked with this library. This is typically the case for commercial programs that are distributed as statically linked executables stripped of any linkage information. Because of the missing symbol table it is impossible to patch the executable binary on disk producing a PBEAM-ready version of it before run time. Therefore, the functionality we previously had in the modified system library and its interface to the PBEAM demon (`pbeamd`) is integrated into a separate System Call Interposition Process SCIP (fig. 3).

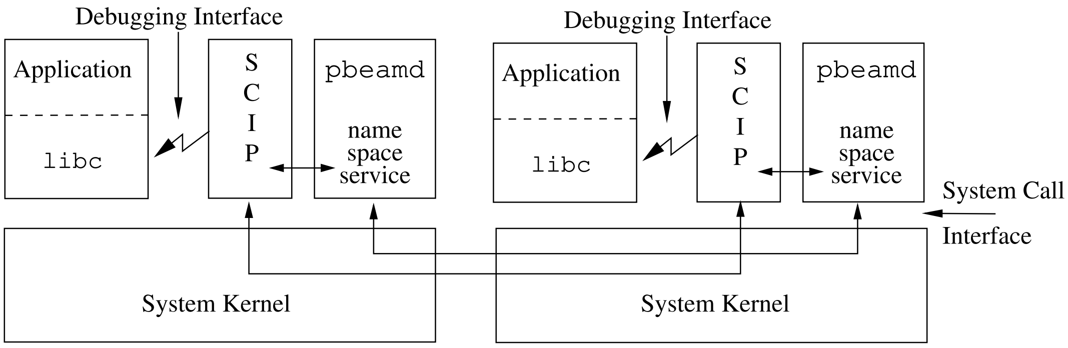

Figure 3 The SCIP in PBEAM.

The debugging interface offers services to access the internal state of another process and to control its execution. We exploit this to transparently let applications run in our virtual name space. The original Unix debugging interface `ptrace()` only allows the inferior process to be halted at breakpoints, continued, and the memory contents be read or written word-wise. These services are sufficient to save the internal state for checkpointing, although they are inefficient. They are insufficient for performing the described name space translations. However, in most modern Unix versions the debugging interface was extended by services to access entire ranges of the inferior's memory, and to let it stop at entry and exit of any or selected system calls. Also they offer extensions that can help to overcome the performance bottlenecks that have been reported in previous work for similar approaches [MS88]. These extensions range from enriched `ptrace()` variants to different kinds of process file systems and combinations of both [Sun90, FG91].

In the following subsections, after a short discussion of the eligible debugging interfaces, we take a closer look at the problems raised by the above mentioned three tasks of the SCIP, and their possible solutions.

3.1 Controlling Another Process

Since the application program – at least in the beginning – contains neither code for exchanging information with its control process nor for being controlled, SCIP depends on the services the system kernel provides for debugging. The only debugging facility found on all Unix systems is the `ptrace()` system call. But the standard Unix `ptrace()` facility offers only minimal support for breakpoint debugging. So using the `ptrace()` interface for SCIP is only possible by using machine dependent extensions to this call. This makes the implementation more difficult and less portable than the modified system library. Besides that it imposes bad performance: each `ptrace()` request causes a context switch to the controlled process, while traditionally the amount of data being transfered by one `ptrace()` call is only one word.

Another standardized Unix debugging interface is the `/proc` file system defined in System V Release 4 [FG91]. This virtual file system provides access to another process' address space by means of standard `read()` and `write()` operations and a variety of `ioctl()` requests to control its execution. It promises better performance, more sophisticated control and a good chance of being portable, but a complete implementation of a `/proc` file system is only rarely (if at all) found in the Unix systems in use today.

In the design of the SCIP, we have considered the specifics of SunOS, Solaris, Linux and Irix. Currently, our main implementation platform is a cluster of workstations running SunOS 4.1.x, which does not have a `/proc` file system but several `ptrace()` extensions. Work on Solaris 2.x has just begun.

3.2 System Call Interposition

In our terms, the system call interposition consists of several phases:

1. *Executing the process until the next system call entry.*

2. *Notification of a system call entry*, i. e. getting notified that the process wants to perform a system call. It must be stopped *before the system call is performed*. The system call arguments including the system call number must be read.

3. *Changing the arguments*, i. e. the virtual names among the arguments must be looked up in the name space mappings and translated into the real name space.

4. *Performing the system call.*

5. *Notification of the system call exit* and its return values, i. e. the process has just exited kernel mode.

6. *Changing the return values* again from real to virtual names.

7. *Restarting the last system call* if necessary.

As noted before, steps 2, 4 and 5 are supported by most modern Unix flavors. SunOS offers a special `ptrace()` request indicating each system call entry and exit. The `/proc` file system even allows for naming a set of system calls to be observed, leaving uninteresting calls untouched. The popular debugging tools `strace` and `truss` are based on this feature. The important difference between SCIP and a tool that just wants to show which system calls are executed with which arguments and results, is that the former also wants to perform the system call with changed arguments, or even not to perform it at all. This is a task the debugging interfaces obviously were not designed for.

On SunOS, when SCIP gets notified about a system call entry, the arguments have already been copied from user address space to kernel memory. They can be read before continuing but not be changed. Also there is no way of forcing the process to leave kernel mode without performing the system call. The problem of changing the arguments in kernel memory probably could be overcome by writing to kernel virtual memory, but having write permissions to kernel memory opens too many security risks to be feasible. The implementation via the `/proc` file system does not encounter this problem.

On the other hand, changing the results after the process has left kernel mode simply works as one would assume. Also, if an argument in fact is a pointer to a memory area in user address space holding the actual data, then the pointer cannot be changed, but the memory contents can. This is true, e. g., for socket addresses.

Another problem with the `ptrace()` paradigm is that stopping the traced process on system call entry or exit happens via delivery of a SIGTRAP signal. So the process must be prevented from blocking this signal, respectively to keep in control if it did manage to block it. The `/proc` file system does not depend on the delivery of a signal, so it is easier to guarantee fail safe tracing.

For these reasons the implementation of SCIP via `ptrace()` cannot be done straight forward, but requires careful distinction between several classes of system calls. In the simplest case the arguments can (or do not have to) be changed in user space, the traced process may safely perform the system call and the return value is adjusted. Examples in this class are `getpid()`, `gettimeofday()` and `sigvec()`.

Other calls, like `sigblock()`, cause effects that can and must be changed or reverted afterwards, in most cases by placing some code into the inferior's address space and forcing it to execute this code, thus patching the application at run time.

Things get even more difficult with non-idempotent calls causing unwanted effects that cannot be reverted, e. g. most I/O operations. Here we must be a bit inventive. In case of I/O operations, successful execution can be prevented by immediately closing any file (or socket) descriptor each time one is opened by the application – using a call to `close()` patched into the application. Subsequent operations on this descriptor will fail, giving SCIP the opportunity to perform the actions with translated arguments. Because doing I/O operations on the SCIP side involves transferring relative large amounts of data between SCIP and the inferior, we will discuss several optimizations along with the performance figures in section 4.

If the application forks a child process, also a new SCIP is spawned that attaches itself to the application's child. When using `ptrace()`, the child must be prevented from running before it can be attached by means of a breakpoint instruction. When using `/proc`, it is stopped automatically. After attaching, the new SCIP can register as a new client to the `pbeamd`.

3.3 Freezing the Application

After the PBEAM demon has requested SCIP to initiate a checkpoint or migration, the traced process must be stopped, if it is not already stopped in user mode. This may be done via `/proc` or by simply sending a SIGSTOP. If the process is in kernel mode, i. e. a system call is in progress, that must be finished first, possibly interrupting a blocking call like `sigpause()`. Such an unintended interrupted system call must be repeated when restarting the checkpoint. So the set of registers, including program counter and stack pointer, that are to be saved are either the current or those from before starting the interrupted call.

Next all the external state information must be read that was not already recorded by the interposition of system calls, e. g. the contents of the interval timers, the set of pending signals and installed signal handlers. Only some of this could be read from the process' user page using a `ptrace()` extension, so SCIP again uses a relocatable compiled function that the traced process is forced to execute.

At last the process' address space must be saved, i. e. written into a file descriptor. One possibility is to copy the whole address space piece by piece into SCIP's address space, then write it. Even if there is a `ptrace()` extension as in SunOS for transferring a large piece of memory with only one call, there is a remarkable overhead, which should be avoided – especially if we consider that migration is most probably initiated when the current machine is overloaded.

A more complex, but faster alternative is to use the Unix file descriptor passing feature. SCIP passes the access rights to its checkpoint/migration file descriptor to a function again patched into the inferior process. This can then efficiently write the whole address space down from "inside" without any detouring.

A significant part of information about the external state is contained in SCIP's own data structures. Therefore, SCIP in fact does a checkpoint of, respective migrates, itself using the checkpoint mechanism provided by our modified system library. In the middle of freezing itself, it freezes the application state and hides the second process image in its own checkpoint.

To save a checkpoint, a new executable file is built from the text and data parts of the process, while the stack contents are written into a separate file. The files are written either locally or via TCP to a server process. In case of migration, the server process does not write the data to files, but overlays its own address space with the executable image sent on TCP, and then passes control to the restart mechanism.

3.4 Restarting the Application

In both cases, whether restarting from a checkpoint executable or after migration, the first step is to revive the saved SCIP image on the new host, which then in turn restarts the application process from the data contained in its address space image. To restore the application's stack contents, a function containing a call to the `read()` kernel service is patched into the running process.

Thus when restoring the application's process image no (slow) memory transfer via the debugging interface is necessary. This is rather important because the `ptrace()` extension in the used SunOS version that should write larger pieces of memory from the controlling to the controlled process does not work correctly, and the alternative of poking the stack image with `ptrace()` word by word into place has shown to be unacceptably slow ($>$ 10 min / 8 MB on Sparc IPC).

After the application is restarted, the SCIP restores the I/O state, that is open files and network connections. Once more the signal status and the interval timers are restored using a relocatable function patched into the application. Signals that had been sent to the process before checkpointing but had not been delivered because they were blocked, are resent to the process with `kill()`.

After SCIP has restored the saved set of registers via the debugging interface, it continues with controlling the application exactly where it had been interrupted.

Note that, while all that function code, that we patch into the controlled process, is of course machine dependent, the *functionality* of them is not. In order to construct them merely a compiler that can produce relocatable code (e. g. GCC) and several assembler instructions encapsulated in preprocessor macros are needed. In fact, the use of code patching instead of accessing kernel data structures decreases porting effort. The architecture, operation principle and most of the code of SCIP can be kept when porting.

4 Performance Figures

In this section we summarize performance for interposing selected system calls and the costs of migration/checkpointing.

The main reason for doing migration and checkpointing is saving time. So it is vitally important that the overhead caused by the mechanisms for interposing system calls, providing the virtual name space as well as freezing, transmitting and restoring a process image, does not compensate all the time saved, e. g. by migration from an overloaded machine to an idle one.

Therefore we will show some performance figures regarding the costs of interposing selected system calls and the costs of migration/checkpointing. As noted before, the platform is a network of different Sun workstations running SunOS 4.1.4, connected by a 10 MBit/s Ethernet. The Sparc 20 is a fast model, Sparc 2 and IPC are slower, and the diskless SLC is the slowest one. Since there is no `/proc` file system, the traditional `ptrace()` interface plus the SunOS-specific extensions are used.

All figures show the averaged data of 5 repetitions of the benchmark runs, done at night on an idle but not dedicated cluster. We used small self-made programs to measure the various aspects of interposing system calls during normal program execution, and of migration respectively checkpointing.

4.1 Costs for Interposing System Calls

There is no constant factor by which the real execution time of an application could be multiplied when running with SCIP and/or the virtual name space. The degree an application is slowed down by the interposition depends heavily on the number of system calls it does. If it does almost no system calls (e. g. only in the beginning and in the end) or if it sets itself asleep frequently, there will be almost no slow down. If it does a lot of system calls, the slow down still depends on the types of the calls.

We have measured the costs for doing system calls when the application is running in the real name space (without any system call interposition), with the modified system library ("libmod"), and with SCIP.

4.1.1 Calls Without Interaction with the Name Space

Times for such system calls when interposed by SCIP are between 0.002 and 0.02 seconds, depending on the complexity of the interposition and the CPU speed. In all cases they are much longer than the times without interposition (factor 50. . . 100)[Bol97]. The corresponding times without any interposition or with interposition through the modified system library were identical and almost zero, because no interposition and no communication with the `pbeamd` is needed in these cases. However, even in the worst case the times are still short enough to be neglected, assuming a system call is a rather rare event. Figure 4 shows the execution times of some selected system/library calls measured on four different Sun models. For each experiment and machine type, we show times for executing the application in the real name space, without any interposition component (curves labeled "real"), for execution in the virtual name space using the modified system call library ("libmod"), respectively interposed by SCIP ("SCIP"). The tests are sorted by decreasing complexity of the actions SCIP must perform in response to the call.

signalTRAP The application tries to use the `signal(3)` function to install a signal handler for the debugging trap signal SIGTRAP. From SCIP's view this results in a sequence of calls to `sigblock()` for blocking SIGTRAP, `sigvec()` for installing the handler and `sigsetmask()` for restoring the previous signal mask. At first SCIP must override the blockage as described for "blockTRAP" below. Then the `sigvec` structure must be changed to prevent the installation of the handler.

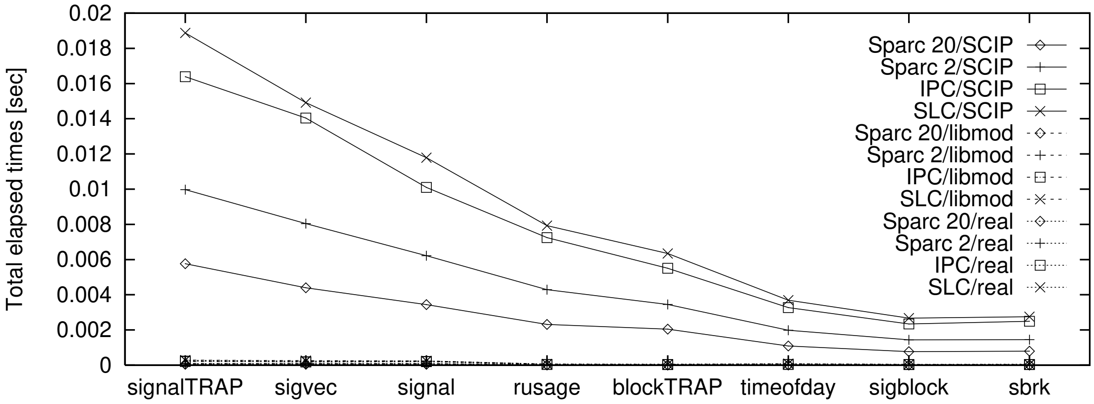

Figure 4 Execution times for selected system/library calls.

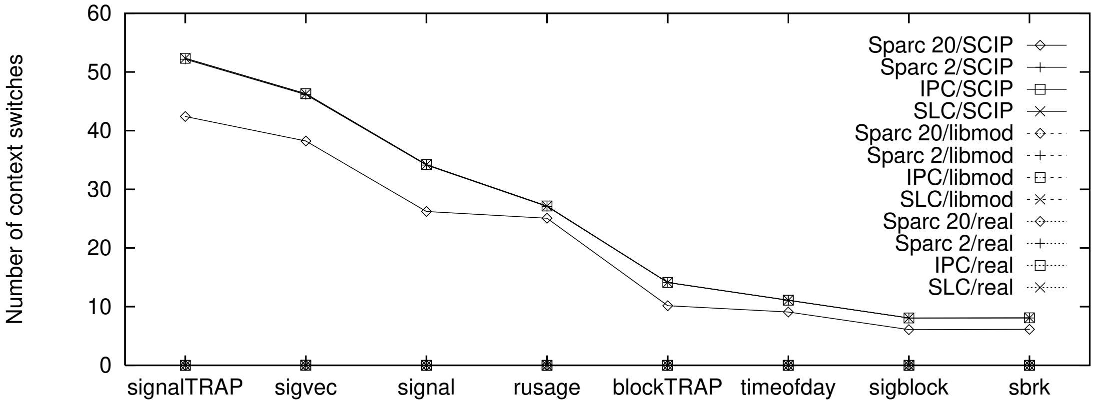

Figure 5 Number of context switches.

sigvec The `sigvec()` call allows to specify a signal mask that is automatically installed when the handler is called. If this signal mask contains the trap signal – like in this test – SCIP must remove it, i. e. again change the `sigvec` structure.

signal For comparison, the application now uses `signal(3)` to install a signal handler that does not harm the trap signal. SCIP must check the `sigvec` structure, but not change it.

rusage If the application requests its resource usage by `getrusage(RUSAGE_SELF, ...)`, SCIP adds the application's and SCIP's own usage of resources and changes the returned structure.

blockTRAP The application adds the TRAP signal to the mask of blocked signals. SCIP cannot prevent this, so a breakpoint instruction must be set just behind the call. When the application runs into the breakpoint, the blockage is reset and SCIP keeps in control.

timeofday If the application reads the system time, any intervals in which the process was frozen, must be hidden; so SCIP needs to subtract a delta value from the actual system time and change the returned time value.

sigblock The application does a "harmless" call to `sigblock()`, for comparison to "blockTRAP". After SCIP has checked that SIGTRAP is not going to be blocked, no further action is required.

sbrk The `sbrk()` is used for allocating or freeing memory. It is an example of a system call that necessitates no special treatment at all.

Figures 4 and 5 show a column for each system call, and in each column a data point for each combination of interposition component and machine type. Note that the lines do not denote contiguous curves but are given just to connect corresponding points.

In figure 4 we can see that the execution times under SCIP in all cases are much longer than the real ones. (The "real" and "libmod" times for these tests are identical and almost zero, because no interposition and no communication with the `pbeamd` is needed in these cases.) Not surprisingly, the slow down increases with decreasing processor speed, but in the worst case the times are still short enough to be neglected, assuming a system call is a rather rare event.

One main cause for the overhead imposed by interposing via the debugging interface is the number of additional context switches between controlling and controlled process. Thus, in figure 5 we also show the number of context switches that occurred during the experiments. It shows that this number for each test is almost the same on all machines and is an important performance factor. So all unnecessary `ptrace()` requests should be avoided (see also below, figure 10).

4.1.2 Calls Requiring Interaction with the Name Space Service

Now we look at some system calls that require interaction with the virtual name space administration in the PBEAM demon.

Figure 6 shows times for querying a process' own identifier (`getpid()`), and that of its parent process (`getppid()`), measured on two machine types. The process' own virtual PID remains constant during its life time, while the parent's virtual PID may change (to 1, when the parent exits) and thus must be looked up from the name space service. The column labeled "SCIP/pbeamd" shows the times for the interposition and, if needed, lookup operation. In contrast, the column simply labeled "SCIP" shows the times for just interposing the system call *without* interaction with the `pbeamd`. The "libmod" and "real" curves again show the numbers for using the other respectively no interposition component.

It also shows that, because the C library functions `getpid()` and `getppid()` under SunOS are implemented using the same system call, SCIP cannot distinguish them and must always perform the expensive action of looking up the parent ID.

The most expensive system call under SCIP is creating a new process with `fork()`, since it requires forking two processes, attaching them and registering to the name space. Figure 7 shows the times needed for this, again for two different machine types and the four different kinds of interposition.

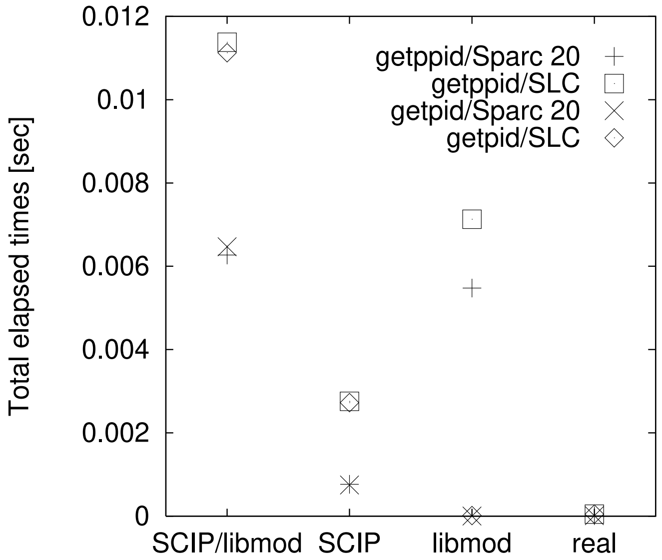

Figure 6 Times for getting the process identifier and the parent process identifier.

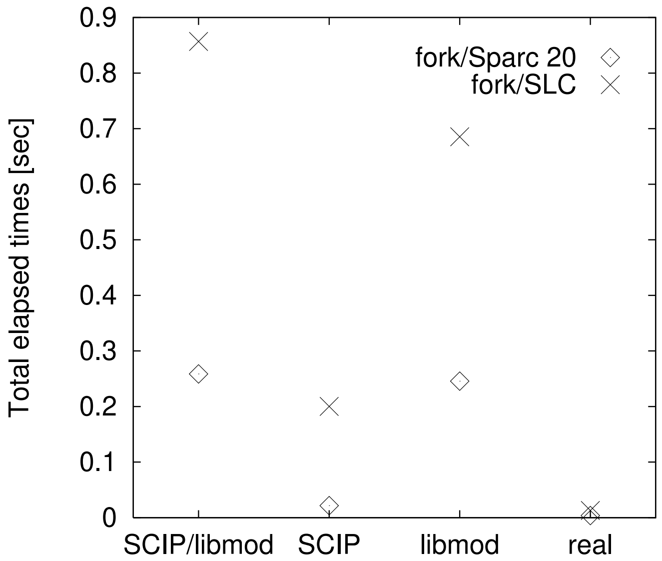

Figure 7 Times for forking a child process.

4.1.3 Communicating Processes

To handle parallel and distributed applications, it is important to support the system calls for I/O and inter-process communication.

Special attention must be payed to I/O operations. In contrast to other system calls, whose arguments are either integers or structures of only several words, the data area to be read or written by an I/O operation is typically much larger – regarding streams, in theory, of almost arbitrary size. Nonetheless they must be considered as arguments respectively return values.

One simple approach would be to let the controlled application perform these op-

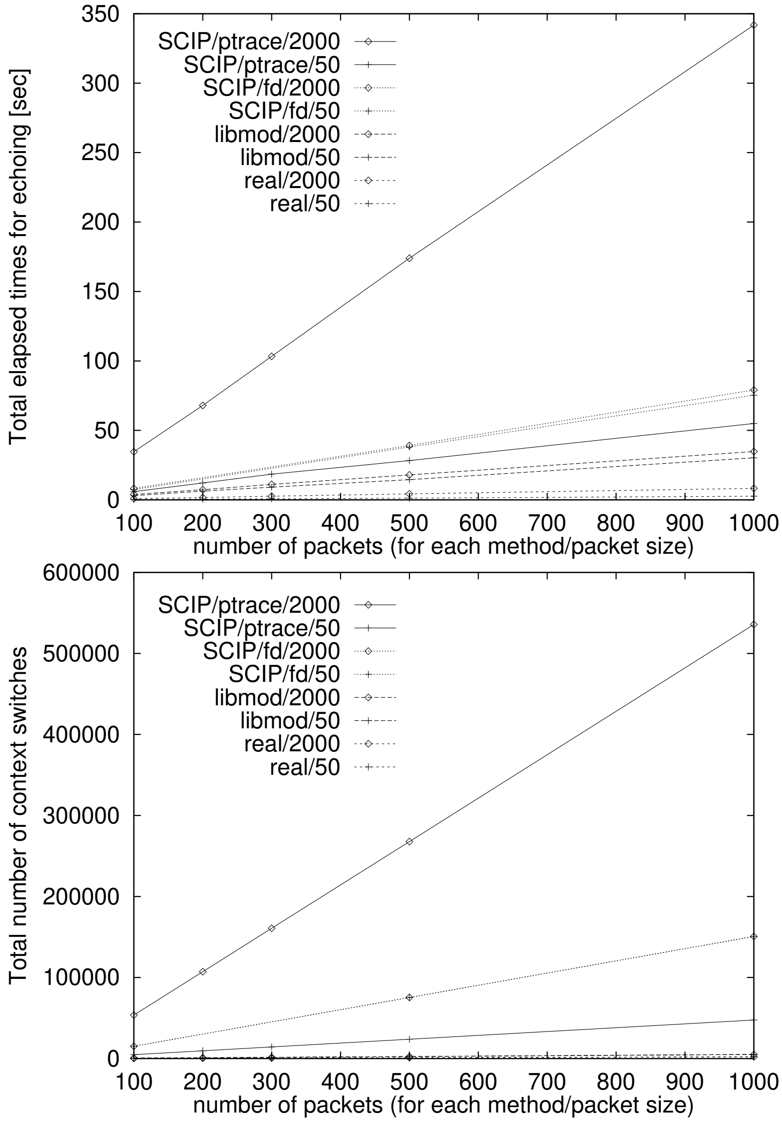

Figure 8 Total elapsed times (above) and number of context switches (below) needed for echoing packets between two Sun IPC workstations.

erations, if necessary after translating any virtual to real addresses. Unfortunately, this does not work for operations that have to conform to a certain protocol (and even cache data) in order to cope with migration and global checkpointing of distributed applications [PSLS96], i. e. especially for interprocess communication.

So another simple approach would be to let SCIP maintain these file descriptors and perform these operation on behalf of the application. This involves exchanging all the data read or written between SCIP and the application.

Figures 8 and 10 show times and number of context switches for a “pingpong” benchmark program. In these tests, two Sun IPCs are echoing back and forth a certain number of messages using UDP datagrams, without working on these messages. This is not a typical behavior of distributed applications, but a good example to show the cost of in-

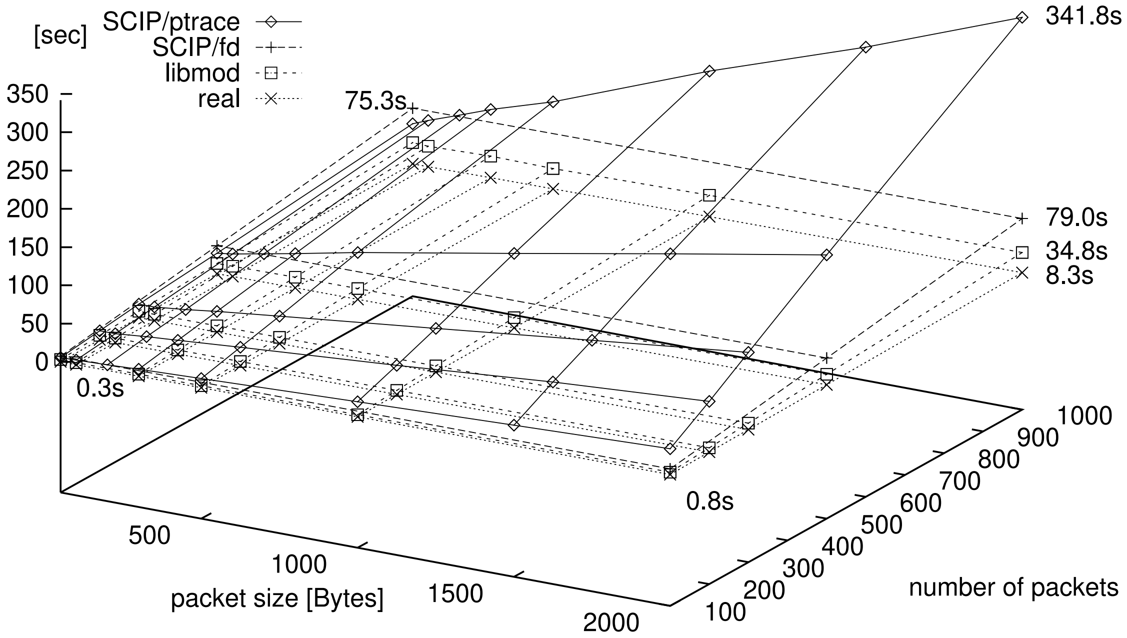

Figure 9 The 3-D view once again shows the small advantage of the "pure" `ptrace()` version over file descriptor passing for small packets (at the left edge), but also the big disadvantage with increasing packet size.

terposing the communication operations. The `sendto()` and `recvfrom()` operations used in the "pingpong" require to look up the real destination resp. virtual sender address in the name space service. They can send and receive messages of variable size. The figures show the times for executing the benchmark with two different packet sizes (50 and 2000 Bytes), and with four different interposition methods. Figure 9 repeats the data from the left side of figure 8 in a 3-dimensional view.

The curve labeled "SCIP/ptrace" shows the times when SCIP performs all communication on behalf of the application process. This is very expensive because the received messages must be poked word by word into pingpong's target buffer.

To overcome this overhead, we have implemented two optimized versions of SCIP. The first one ("SCIP/fd") avoids copying the received message data by passing the access rights to its own file descriptor to the pingpong program, that can then receive the message on its own. This involves patching and executing a short function in the application each time `recvfrom()` is called. For large packets the time savings compared to the `ptrace()` version are immense, but poking this function again and again onto the application's stack is expensive, too. For small packets this even causes a small disadvantage compared to the `ptrace()` version.

Therefore, the second experimental version ("SCIP/fd/shared mem" in figure 10) puts this function into a shared memory segment, so that it can at any time be called without any overhead. This again saves most context switches and almost half the time. The measured data shows that if – and only if – such mechanisms are used where possible, the performance of SCIP can get close to that of the modified system library, that does not

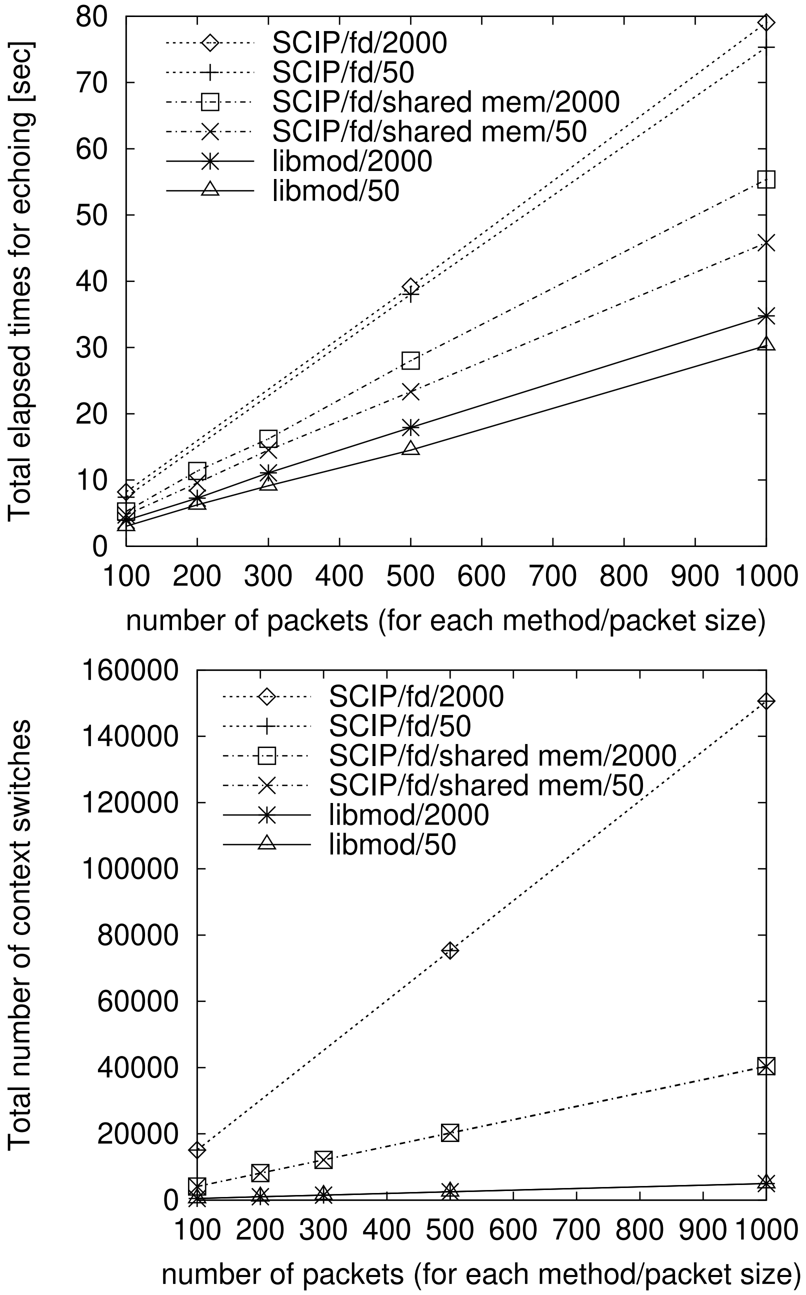

Figure 10 Results of the pingpong benchmark: using Shared Memory saves almost half the time (above) and most context switches (below).

encounter any problems with involuntary context switches. The advantage seems worth the more difficult implementation.

4.2 Costs for TCP Connections

PBEAM does not only support UDP communication but also applications that use TCP

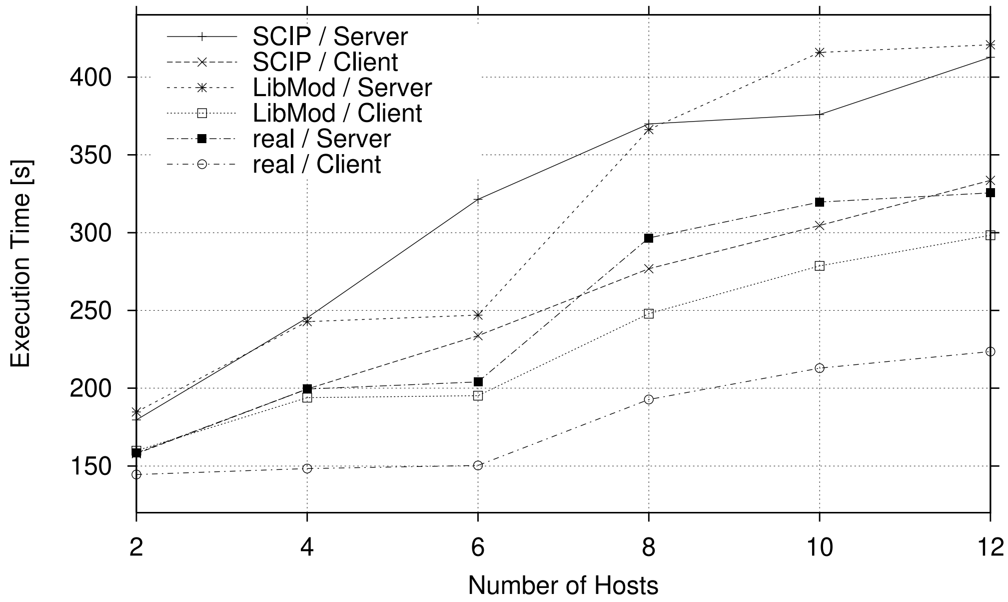

Figure 11 Total elapsed times for exchanging small packets between one server and up to eleven clients via TCP connections.

connections. In this case interaction with the name space is only necessary in order to connect two sockets. After the connection is set up, data can be transmitted as usual until the application is frozen for a checkpoint or migration. Before the connection is closed cleanly, both sides flush any pending data into special buffers to prevent data loss. As soon as the TCP connection is used again for reading or writing after migration/rollback, it is re-connected with help of the name space administration. If there was data saved in the special buffers, all further input/output must first be redirected via these buffers, i. e. the real read or write system calls must be delayed, until these buffers are empty again. The consequences of this design are that setting up a TCP connection is very expensive while the normal overhead for using the connection is much lower than for UDP communication.

The costs of migratable TCP connections have been measured with the “tcpong” benchmark program which was derived from “pingpong”. One server program accepts TCP connections from clients and echoes all data received data back to the sender. To test the scalability of our approach, up to eleven clients have been used in our tests. Figure 11 shows the total elapsed times for sending and echoing 100 packets à 2000 bytes per client. For each version of the program the averaged times of the server and the client program are shown respectively. The slow down introduced by SCIP ranges between factors 1.2 and 1.6 while the modified system library causes a slow down of 1.2 to 1.4. In our opinion this result is very encouraging.

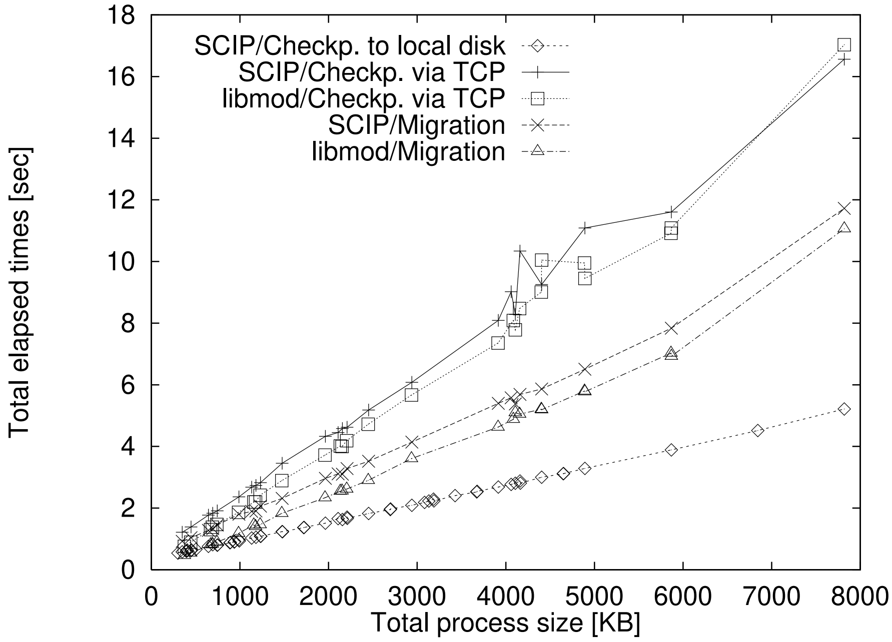

Figure 12 Times for checkpointing/migration between two Sun IPCs.

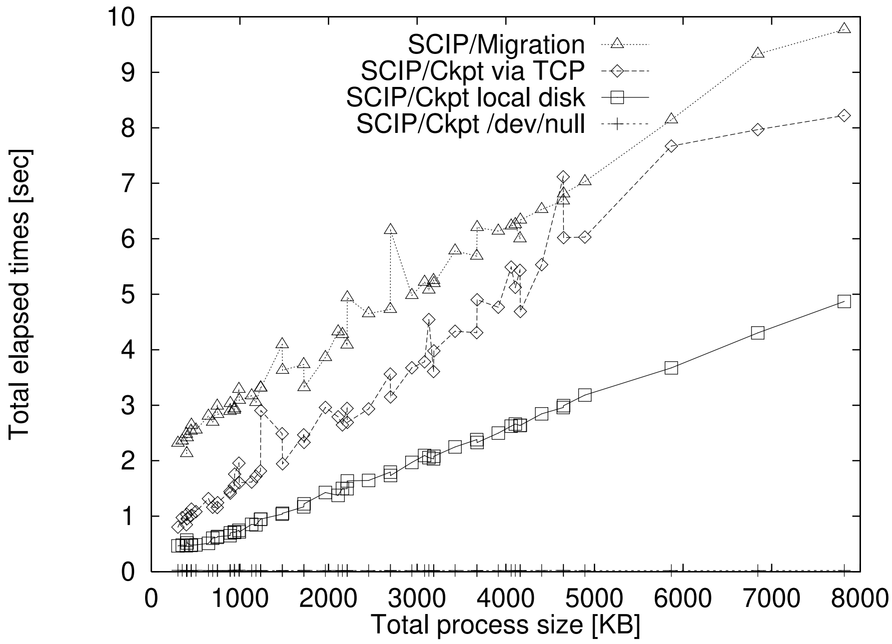

Figure 13 Times for checkpointing/migration from a SPARCstation 10 to a SPARCstation 20.

4.3 Costs for Migration and Checkpointing

Figures 12 and 13 show the total elapsed times for checkpointing and migrating processes of varying size up to eight MB of data and stack memory with SCIP. The state image is either written to a local disk file or sent via TCP to a server process running on the target machine and written to the local disk there. This is much faster than using NFS. Migration is likewise done via TCP, as described in section 3.3.

In the figure 12 both source and target machines are Sun IPC workstations. Checkpointing and migration are done using SCIP as well as the modified system library ("libmod"). As described in sections 3.3 and 3.4, the same mechanisms for reading and writing the process image are used by both methods, and the measured data confirms that both take almost the same time, with SCIP causing only a little and constant overhead. Here, the bottleneck was a slow hard disk and very low memory on the server machine. With better equipped machines we also have seen the network become the bottleneck [PL95].

Figure 13 shows the times for checkpointing and migration from a SPARCstation 10 to a SPARCstation 20 with a faster hard disk showing the checkpoint/migration speed that is reachable over a 10 MBit/s Ethernet. For comparison, there are also two curves of checkpointing to the null device. The current implementation of SCIP does not spend any time on this, but a former version that copied the process image page-wise using `ptrace()` needed some notable time for this.

4.4 Performance Summary

From the experiences with our modified library approach, we have seen that, although the overhead for doing single system calls is high, typical long-running scientific applications do a significant amount of computation between system calls, so that the overall overhead for the execution in the virtual name space is lower than the deviation of the measured times in repeated benchmark runs ($1 \ldots 5\%$).

Also, we note that the migration costs are roughly in the same order of magnitude that is reported about migration mechanisms in "real" distributed operating systems (see e. g. [Mil94, PL95]).

5 Related Work

In the last years, several process migration and checkpointing systems have been developed, which run outside the system kernel in user space only.

Condor [LS92, LTBL97] provides migration and rollback transparency for sequential applications through a modified system library that forwards the system call arguments back to a shadow process on the application's home node to invoke the real service there. It does not support signals, timers or communication operations, which are needed for distributed applications.

Mandelberg and Sunderam [MS88] read the internal state via `ptrace()` for creating checkpoints. Information about the external state is read from kernel memory. They do not use a modified system library, thus they can handle unmodified binary programs. Applications are allowed to access files on NFS-mounted file systems only. They do not interpose system calls but collect information from kernel data structures and create links to the files by faking NFS link calls. They report that migration is slow because for one reading the address space with the "traditional" `ptrace()` facility is expensive, and because they transferred the state information via NFS. They need 3 seconds per 100 KB process image size on Sun3 machines, while SCIP achieves about 0.12…0.22 s / 100 KB.

In the ARTEMIS project [SHSK97], the operating system's shared library mechanism is exploited to link commercial applications with a modified system library. This, obviously, requires the applications to be dynamically linked, and is not applicable to statically linked programs. In PBEAM and SCIP we did not yet implement support for shared memory and thus not for dynamically linked applications.

Other migration systems require the applications to be based on some specific parallel programming environment like PVM [GBD$^+$94] or MPI [MPI97]. They provide transparency for the objects' names (e.g. task IDs) that are defined by the programming environment. They all require the applications to be linked with special versions of the parallel programming library [Ste96, CCK$^+$95, LFS93, VHOS95, AEFS93]. From PBEAM's point of view, these programming environments are part of the application.

Trinitis is integrating an external checkpointer into CoCheck [Tri98, Ste96] to do the single process checkpointing. It also controls the application via the debugging interface. However, to provide migration transparency beyond the internal state, CoCheck currently uses wrappers for the message passing library. Here, PBEAM/SCIP has the advantage of working at the system call level. The debugging interface does not directly support interposing the message passing library operations. To do that via the debugging interface would require to have access to the symbol table of the application, so the control process could place breakpoints at the message passing library functions, and could manipulate the appropriate data structures.

A very similar technique to SCIP is used by Alexandrov et al. for a different purpose [AISS97]. Their Ufo file system extends the operating system functionality at user level with user-installable file systems. The file operation system calls are intercepted via the `/proc` file system and their arguments and return codes changed.

6 Conclusions

We conclude that the separation between the system call interposition component and the administrative components of PBEAM makes it possible to choose that interposition component that is suited best for the specific task at hand. Besides the modified system library we have explored ways to transparently handle arbitrary binaries, that are already linked. The tradeoff, apart from implementation issues, is between transparency and performance. Although the approach via the debugging interface is more difficult to implement than the modified system library, we can still achieve some degree of portability with it (see also the remarks about portability of Ufo [AISS97]).

Nevertheless it should be noted, that porting SCIP to another platform is much more likely to raise problems than porting the modified system library. The latter applies to the system's C library interface, which is usually well specified and does not differ too much between Unix flavors. In contrast, SCIP applies to the system's trap interface which might differ largely from the library interface and is often bad documented, if at all. E. g. on our next target platform Solaris 2.{5,6}, the kernel itself does not know of sockets but uses Streams. This does not matter for the modified system library, since the socket interface is still provided through a special library. New functionlaity must be added to SCIP to handle Streams. There is another new feature at the system call interface in Solaris 2.5, the door mechanism, which is only documented as "not yet available for public consumption".

The presented measurements show that also an acceptable performance almost similar to that of the modified system library approach can be gained. The performance is significantly better than what we initially expected from the numbers reported in the literature for previous work [MS88]. The main reasons for this improvement are the exploitation of the debugging interface extensions of modern Unix flavors, and even more the avoidance of as many context switches between application and interposition process as possible.

References

Many of the cited documents and some more are available online in the Internet. We have set up a World-Wide-Web page with references on `http://www.cs.tu-bs.de/~petri/pbeamrefs.html` and `http://www.iti.mu-luebeck.de/~petri/pbeamrefs.html`.

[AEFS93] Duane Andres, Chris Elford, Brian Fin, and Lincoln Smith. Dynamic load balancing in PVM. Technical report, University of Illinois at Urbanna-Champaign, April 1993.

[AISS97] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. Extending the Operating System at the User Level: the Ufo Global File System. In *USENIX Technical Conference Proceedings*, pages 77–90, Anaheim, CA, January 1997.

[Bol97] Matthias Bolz. Transparent Redirection of System Calls for Unmodified Programs in ⟨PBEAM. Master's thesis, Institut für Betriebssysteme und Rechnerverbund, TU Braunschweig, November 1997. (In German).

[CCK+95] Jeremy Casas, Dan L. Clark, Ravi Konuru, Steve W. Otto, Robert M. Prouty, and Jonathan Walpole. MPVM: A migration transparent version of PVM. *Usenix Computing Systems*, 8(2):171–216, 1995.

[CCS95] CCS Annual Report. WWW page, Center for Computational Sciences, Oak Ridge National Laboratory, 1995. `http://www.ccs.ornl.org/AnRep95/CCS95.html`.

[CM92] J. Cargille and Barton P. Miller. Binary Wrapping: A Technique for Instrumenting Object Code. *ACM Sigplan Notices*, 27(6):17–18, June 1992.

[FG91] Roger Faulkner and Ron Gomes. The Process File System and Process Model in UNIX System V. In *USENIX Technical Conference Proceedings*, pages 243–252, Dallas, TX, January 1991.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, Massachusetts, 1994.

[Jon92] Michael Blair Jones. *Transparently Interposing User Code at the System Interface*. PhD thesis, CMU, September 1992.

[KHG96] Alan H. Karp, Michael Heath, and Al Geist. 1995 Gordon Bell Prize Winners. *IEEE Computer*, 29(1):79–85, January 1996.

[KLB98] Alan H. Karp, Ewing Lusk, and David H. Bailey. 1997 Gordin Bell Prize Winners. *IEEE Computer*, 31(1):86–92, January 1998.

[LCP91] D. Long, J. Caroll, and C. Park. A Study of the Reliability of Internet Sites. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 177–186, 1991.

[LFS93] Juan León, Allan L. Fisher, and Peter Steenkiste. Fail-save PVM: A portable package for distributed programming with Transparent Recovery. Report CMU-CS-93-124, Carnegie Mellon University, Pittsburgh, PA, February 1993.

[LS92] Michael J. Litzkow and Marvin Solomon. Supporting Checkpointing and Process Migration Outside the UNIX Kernel. In *USENIX Technical Conference Proceedings*, pages 283–290, San Francisco, CA, January 1992.

[LTBL97] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpointing and Migration of UNIX Processes in the Condor Distributed Processing System. Report 1346, University of Wisconsin-Madison Computer Sciences, April 1997.

[Mil94] Dejan S. Milojičić. *Load Distribution – Implementation for the Mach Microkernel*. Vieweg Verlag, Braunschweig, 1994.

[MPI97] Message Passing Interface Forum MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, July 1997. `http://www.mpi-forum.org`.

[MS88] K.I. Mandelberg and V.S. Sunderam. Process Migration in UNIX Networks. In *USENIX Technical Conference Proceedings*, pages 357–363, Dallas, TX, February 1988.

[PBL98] Stefan Petri, Matthias Bolz, and Horst Langendörfer. Migration and Rollback Transparency for arbitrary Distributed Applications in Workstation Clusters. In José Rolim, editor, *Parallel and Distributed Processing – 10 IPPS/SDPD'98 Workshops held in conjunction with the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing, Orlando, Florida, USA; Proceeedings*, volume 1388 of *Lecture Notes in Computer Science*, pages 159–170, Berlin, March 1998. Springer.

[PL95] 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.

[PSB+97] Stefan Petri, Bettina Schnor, Matthias Becker, Bernd Hinrichs, Titus Tscharntke, and Horst Langendörfer. Evaluation of Multicast Methods to Maintain a Global Name Space for Transparent Process Migration in Workstation Clusters. In Martina Zitterbart, editor, *Kommunikation in Verteilten Systemen*, pages 224–234. GI/ITG Fachtagung KIVS'97, Springer, February 1997.

[PSLS96] Stefan Petri, Bettina Schnor, Horst Langendörfer, and Jens Steinborn. Consistent Global Checkpoints for Distributed Applications on Clusters of Unix Workstations. In Hermann G. Matthies and Josef Schüle, editors, *Paralleles und Verteiltes Rechnen – Beiträge zum 4. Workshop über Wissenschaftliches Rechnen*, pages 77–86, Aachen, October 1996. TU Braunschweig, Shaker.

[SHSK97] Toshio Shirakihara, Hideaki Hirayama, Kiyoko Sato, and Tatsunori Kanai. ARTEMIS: Advanced Reliable disTributed Environment Middleware System. In

Hamid R. Arabnia, C. Ierotheou, Ronald A. Olsson, Yi Pan, R. Pandrey, and E.G. Talbi, editors, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'97*, pages 97–106, Las Vegas, NV, July 1997. CSREA.

[Ste96] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, April 1996.

[Sun90] Sun Microsystems. *SunOS Reference Manual*, 1990. Revision A.

[Tri98] Jörg Trinitis. An External Checkpointing Technique for Integration into a Parallel Tool Environment. In preparation. `trinitis@informatik.tu-muenchen.de`, 1998.

[VHOS95] Joep J.J. Vesseur, Robbert N. Heederik, Benno J. Overeinder, and Peter M.A. Sloot. Experiments in Dynamic Load Balancing for Parallel Cluster Computing. In Peter Fritzon and Leif Finmo, editors, *Proceedings of the Workshop on Parallel Programming and Computation (ZEUS'95) and the 4th Nordic Transputer Conference (NTUG'95)*, pages 189–194, Amsterdam, June 1995. IOS Press.

Technische Universität Braunschweig
Informatik-Berichte ab Nr. 94-09

95-01	V. S. Cherniavsky	Philosophische Aspekte des Unvollständigkeitstheorems von Gödel
95-02	G. Snelting	Reengineering of Configurations Based on Mathematical Concept Analysis
95-03	A. Zeller	A Unified Configuration Management Model
95-04	H. Bickel, W. Struckmann	The Hoare Logic of Data Types
95-05	F.-J. Grosch	No Type Stamps and No Structure Stamps – a Referentially-Transparent Higher-Order Module Language
95-06	V. S. Cherniavsky	Über semantische und formalistische Beweismethoden in den exakten Wissenschaften
95-07	A. Zeller, D. Lütkehaus	DDD - A Free Graphical Front-End for UNIX Debuggers
95-08	A. Zeller	Smooth Operations with Square Operators – The Version Set Model in ICE
95-09	P. Funk, A. Lewien, G. Snelting	Algorithms for Concept Lattice Decomposition and their Application
96-01	A. Zeller, G. Snelting	Unified Versioning Through Feature Logic
96-02	M. Goldapp, U. Grottker, G. Snelting	Validierung softwaregesteuerter Meßsysteme durch Program Slicing und Constraint Solving
96-03	C. Lindig, G. Snelting	Modularization of Legacy Code Based on Mathematical Concept Analysis
96-04	J. Adámek, J. Koslowski, V. Pollara, W. Struckmann	Workshop Domains II (Proceedings)
96-05	F.-J. Grosch	A Syntactic Approach to Structure Generativity
96-06	E. H. A. Gerbracht, W. Struckmann	Zur Diskussion elementarer Funktionen aus algorithmischer Sicht
97-07	H.-D. Ehrich	Object Specification
97-01	A. Zeller	Versioning Software Systems through Concept Descriptions
97-02	K. Neumann, R. Müller	Implementierung von Assertions durch Oracle7-Trigger
97-03	G. Denker, P. Hartel	TROLL – An Object Oriented Formal Method for Distributed Information System Design: Syntax and Pragmatics
97-04	F.-J. Grosch	M - eine typisierte, funktionale Sprache für das Programmieren-im-Grossen
97-05	J. Küster Filipe	Putting Synchronous and Asynchronous Object Modules together: an Event-Based Model for Concurrent Composition
97-06	J. Küster Filipe	A categorical Hiding Mechanism for Concurrent Object Systems
97-07	G. Snelting, U. Grottker, M. Goldapp	VALSOFT Abschlussbericht
98-01	J. Krinke, G. Snelting	Validation of Measurement Software as an application of Slicing and Constraint Solving
98-02	S. Petri, M. Bolz, H. Langendörfer	Transparent Migration and Rollback for Unmodified Applications in Workstation Clusters