A Survey of Process Migration Mechanisms

Ramon Lawrence
Department of Computer Science
University of Manitoba
umlawren@cs.umanitoba.ca

May 29, 1998

1 Introduction

The days of supercomputers and mainframes dominating computing are over. With the cost benefits of the mass production of smaller machines, the majority of today's computing power exists in the form of PCs or workstations, with more powerful machines performing more specialized tasks. Even with increased computer power and availability, some tasks require more resources. Load balancing and process migration allocate processes to interconnected workstations on a network to better take advantage of available resources.

This paper presents a survey of process migration mechanisms. There have been other such survey papers on process migration[13, 17], but this paper is more current and includes one new algorithm, the Freeze Free algorithm. First, motivations for load balancing and process migration are presented in Section 2. Section 3 gives a brief overview of load balancing. The main contribution of the paper is in Section 4 which describes process migration. The differences between process migration and load balancing are discussed, and several of the current process migration implementations and algorithms are examined. Section 5 presents future work, possible improvements, and extended applications of process migration. Finally, concluding remarks are given in Section 6.

2 Motivations

Load balancing and process migration are used in a network of workstations (NOW) to share the load of heavily loaded processors with more lightly loaded processors. The major reason for load balancing is that there is a lot of distributed processing power that goes unused. Instead of overloading a few machines with a lot of tasks, it makes sense to allocate some of these tasks to under-used machines to increase performance. This allows for tasks to be completed quickly, and possibly in parallel, without affecting the performance of other users on other machines.

There are also more subtle motivations for performing process migration. By moving a process, it may be possible to improve communication performance. For example, if a process is accessing a huge remote data source, it may be more efficient to move the process to the data source instead of moving the data to the process. This results in less communication and delay and makes more efficient use of network resources. Process migration could also be used to move communicating processes closer together.

Other motivations for process migration include providing higher availability, simplifying reconfiguration, and utilizing special machine capabilities. Higher availability can be achieved by migrating a process, presumably a long-running process, from a machine that is prone to failure or is about to be disconnected from the network. Thus, the process will continue to be available even after machine failure or disconnection. This availability is desirable in reconfiguration where certain processes are required to be continually operational (e.g. name servers). Migration allows the process to be available almost continually, except for migration time, as opposed to restarting the process after the service is moved to another machine. Finally, certain machines may have better capabilities (e.g. fast floating point processing) which can be utilized by moving the process to the machine. In this case, the process can be started on a regular machine and then migrated to a more powerful machine when it becomes available.

Process migration may also find application in mobile computing and wide area computing. In mobile computing, it may be desirable to migrate a process to and from the base station supporting the mobile unit. In this way, the process can optimize its performance based on network conditions and the relative processing power of the mobile and base station. Also, as a mobile unit migrates, a process associated with the mobile unit running on a base station can migrate with the mobile unit to its new base station. This avoids communication between the base stations to support the mobile unit's process. Process migration also has applications in wide area computing, as processes can be migrated to the location of the data that they use. This is important because despite increasing network bandwidth, network delays are not being reduced significantly. Thus, it may be beneficial to move a process to the location of the data to improve performance.

In total, there are many motivations for load balancing and process migration. The goal of these techniques is to improve performance by distributing tasks to utilize idle processing power. Process migration can also reduce network transmission and provide high availability. The following sections examine how load balancing and process migration achieve these goals.

3 Load Balancing

Load balancing is the static allocation of tasks to processors to increase throughput and overall processor utilization. Load balancing can be done among individual processors in a parallel machine or among interconnected workstations in a network. For this paper, load balancing and process migration mechanisms will operate in a networked workstation environment, but the algorithms proposed may work equally well in a parallel environment. In either case, there is an implicit assumption of cooperation between the processors. That is, a processor must be willing to execute a process for any other.

The key word in the load balancing definition is **static**. Load balancing occurs before process creation. This is different from process migration which is performed after a process has been created and has executed for some time. By moving a process before it is created, load balancing does not have to handle process execution state, the state of process communication, or the run-time environment, which greatly simplifies the task. However, there still remains some issues to be resolved.

The most fundamental issue in load balancing is determining a measure of load. There are many different possible measures of load including: number of tasks in run queue, load average, CPU utilization, I/O utilization, amount of free CPU time/memory, etc., or any combination of the above indicators. Other factors to consider when deciding where to place a process include nearness to resources, processor and operating system capabilities, and specialized hardware/software features. The "best" load measure as determined by empirical studies is "number of tasks in run queue"[16]. This statistic is readily available using current operating system calls, and it is less time-sensitive than many other measures. Even though the number of the tasks in the system is a dynamic measure, the number of tasks changes slower than the CPU or I/O utilizations, for example, which may vary within a task execution.

The time-sensitivity of an indicator is very important because load balancing is performed with stale information. Each site makes load balancing decisions based on its view of the loads at other sites. Since the job mix at each site is continually changing, it is unlikely that every site will have recent information, and the load balancing algorithm must not make bad decisions based on stale or incomplete information. Fortunately, adequate load balancing can be performed without information from all the nodes in the network. In some cases, information on 3 to 5 nodes is sufficient[16].

The decision of when to load balance is also important. Clearly, it does not make much sense to perform load balancing if a task is only going to execute for a short period of time. Load balancing is only useful for long-running tasks with large CPU and I/O requirements. Studies[16] have shown that 98% of processes only use 35% of CPU time while 0.1% of processes use 50% CPU time. It is those 0.1% of processes that have the most to gain by performing load balancing. Also, long running tasks can be identified by examining their past behaviour. This fact does not affect load balancing decisions as no execution information is available, but it does influence process migration decisions. Overall, load balancing is useful for large jobs, but it is often hard to determine which jobs are going to require the most resources prior to execution.

There are numerous proposed load balancing algorithms. Some of these algorithms are topology specific, very theoretical (and hence not practical), or overly simplistic. Since this is not a survey on load balancing algorithms, two general algorithms, bidding algorithms and drafting algorithms, will be briefly discussed to give an idea of the issues involved.

In bidding algorithms, an overloaded processor sends requests to other processors when a new process is created. Processors respond with "bids" which consist of their ability to process the new process. The overloaded process selects the best bid and sends the new process to that location. Bidding algorithms are simple and are only used when a processor is overloaded. Unfortunately, the overloaded processor is doing most of the work in the algorithm.

In drafting algorithms, a lightly loaded processor sends messages to other processors indicating that it could handle more work, and an overloaded processor can then send a new process to a lightly loaded processor. The advantage in this method is that lightly loaded processors are doing most of the work, but in a lightly loaded system, there may be needless communication. Also, both algorithms may give too many processes to a lightly loaded processor because of stale information.

Hybrids of the two methods partition the network and perform a drafting algorithm within a partition and a bidding algorithm between partitions. The hybrid technique reduces some of the problems found in the techniques applied individually.

In summary, load balancing distributes work among processors to achieve better utilization and throughput. A measure of load is easily obtained by examining the number of tasks in the run queue, but any load balancing algorithm must be able to handle stale and incomplete information about current processor loads. There are many load balancing algorithms. The two general algorithms presented, bidding and drafting, can be applied to any network topology without major problems. Unfortunately, load balancing algorithms must somehow determine the resource utilization of a process before it begins. Although allocating a process before execution simplifies the allocation, it means that the load balancing algorithm may misplace processes or fail to properly estimate execution requirements. For this reason, load balancing should be used in conjunction with process migration.

4 Process Migration

Process migration is the movement of a currently executing process to a new processor. Process migration allows the system to take advantage of changes in process execution or network utilization during the execution of a process. Thus, process migration is especially beneficial for large jobs which have varying resource requirements. In fact, as a job gets larger and its resource requirements become more variable, load balancing has less effect because initial conditions at load balancing time may change. A combination of load balancing and process migration is the most beneficial. By providing a process migration facility, a load balancing algorithm has flexibility in its initial allocation as a process can be moved at any time.

Many of the same problems that occur in load balancing also occur in process migration. For example, a useful measure of processor load is needed, and the algorithm still must handle stale and incomplete load information. In addition, process migration must also handle movement of process state, communication migration, and process restart given state information. Also, process migration must be performed quickly to realize an increase in performance. The mean process execution time is 50ms[16], so process migration should be fast. Thus, the minimization of process migration time is very important.

There are several important performance metrics for evaluating a process migration protocol. The **process migration latency time** is the time from issuing the migration request until the process can execute on the new machine. The **message freeze time** is the time that messages are stopped during process migration. Other important times are the excision and insertion times. **Excision** is the proces of extracting process information from the operating system kernel, and **insertion** is the process of inserting process information into the operating system kernel.

Process migration algorithms must handle process immobility. A process is considered immobile if it should not be moved from its current location. Such processes include I/O processes which interact heavily with the user's terminal (e.g. text editor). Up to 70% of processes can be immobile and process migration is still beneficial[16].

This paper will concentrate on an integrated system of networked workstations, although there are various system models for which process migration algorithms have been proposed:

- **autonomous system -** Each computer belongs to one user, is only available when the owner is not using it, and the system evicts outside processes when the owner returns to the machine. (e.g. Condor)
- **integrated system -** A unified system with a set of resources to be maximally utilized. (e.g. Amoeba)
- **massively parallel processor system -** Parallel machines using application-controlled load balancing and optimization for large problems.

Finally, there are 3 important components common to any process migration policy:

- **load balancing policy -** when and where to transfer a process
- **process migration mechanism -** how the system migrates a process from source to destination
- **remote execution -** process execution on a new machine

In the following sections, three types of process migration will be studied: homogeneous, heterogeneous, and distributed shared memory techniques. Homogeneous process migration occurs between machines with the same architecture and operating system. Heterogeneous process migration allows migration between different machines. Distributed shared memory algorithms apply process migration by using shared memory instead of message passing. The major problems in each environment will be introduced along with example algorithms and current implementations. The focus will be on homogeneous process migration as it is the most efficient and the most commercially viable technique.

4.1 Homogeneous Process Migration

Homogeneous process migration involves migrating processes in a homogeneous environment where all systems have the same architecture and operating system but not necessarily the same resources or capabilities. Process migration can be performed either at the user-level or the kernel level. A brief overview of the two techniques and a few systems that implement them are given in the next sections. In following sections, five generic process migration algorithms are presented. Description of some systems currently implementing these process migration algorithms are presented with the algorithms.

4.1.1 User-level Process Migration

User-level process migration techniques support process migration without changing the operating system kernel. User-level migration implementations are easier to develop and maintain but have two common problems:

- They cannot access kernel state which means that they cannot migrate all processes.
- They must cross the kernel/application boundary using kernel requests which are slow and costly.

User-level process migration facilities vary in complexity from the UNIX rsh command which allows static, remote execution of UNIX commands to Condor[12] which allows dynamic migration of processes using checkpointing. Another implementation[11] relies on the cooperation between the process and the migration subsystem to achieve migration. The problem with these implementations is that without kernel access, they are unable to migrate processes with location dependent information and interprocess communication.

4.1.2 Kernel Level Process Migration

Kernel level process migration techniques modify the operating system kernel to make process migration easier and more efficient. Kernel modifications allow the migration process to be done quicker and migrate more types of processes. Unfortunately, many older implementations have high overhead, long freeze times, and still cannot migrate all processes.

Some current implementations include Locus[3, 22], Clouds[6, 7], Amoeba[20], Charlotte[10, 2], V[5, 4], Accent[15], and Sprite[9, 14]. Some of these implementations will be discussed later, but all have the common feature of being prototype distributed operating systems. Each of these implementations uses one of the first four migration algorithms presented later. Process migration in V was revolutionary as it did not transfer the whole address space at migration time. The Accent implementation was important because it represented the first time demand-paging of the address space was used, and the Sprite implementation was also important as it introduced file system support for process migration. A new proposed algorithm called the Freeze Free Algorithm[16] claims better performance than previous implementations and will be discussed in detail.

4.1.3 Homogeneous Process Migration Algorithms

There are five basic algorithms for homogeneous process migration:

- Total Copy Algorithm
- Pre-Copy Algorithm
- Demand Page Algorithm
- File Server Algorithm
- Freeze Free Algorithm

Any process migration algorithm must be able to handle the new host rejecting migration and should restart the algorithm with a new target host. The algorithm must also establish a transfer order of state components and attempt to minimize round trip messages and the total number of messages. The process state information that must be gathered and transmitted includes:

- **Process control information -** priority, state, process ID, parent process ID
- **Execution state -** processor state, registers, stack pointer, program counter, status registers
- **Address space -** the entire virtual memory of the process
- **Messages -** buffered messages, link control information
- **File information -** file descriptors and cached file blocks

When and how this information is transmitted varies between the algorithms. Obviously some of this information, like the address space, could be vary large, so it would be beneficial to avoid transmiting the entire object at one time. Handling messages during process migration is also difficult because a process may still receive messages. The following sections present current process migration algorithms in chronological order.

4.1.4 Total Copy Algorithm

The Total Copy algorithm is the first and most widely used process migration algorithm. It is used in Locus, Amoeba, Charlotte, and others. The idea of the algorithm is to suspend the process, transfer all state information, and then resume the process. The general algorithm is:

- Suspend process at old host.
- Old host sends migration approval request message to new host.
- New host responds to request message with accept or reject.
- If migration is accepted, the old host:
  - Marshals and transfers process state.
  - Ships communication links and buffered messages.
  - Discards clear cache file blocks and transfers file descriptors and dirty file cache blocks.
  - Ships all code, heap, and stack pages.
  - Tells new host to restart process.

Basically, all relevant process information including all virtual memory is transmitted to the new host during migration. Obviously, this method will occur long delays as the whole virtual address space must be sent. This slow transfer time is important if the process is doing IPC because it cannot receive messages in this time. The message sender may then timeout and assume the receiver has failed. The large overhead of transferring everything at once is clearly undesirable. The advantage of the algorithm is its simplicity, and by shipping all information to the new site, it eliminates residual dependencies. **Residual dependencies** exist when process information remains at the old host after migration. A system is fault tolerant only if no residual dependencies exist because a residual dependency implies that the process is still dependent on its former location and will fail if that location fails. Since the first three steps, in which the new host asks for migration permission, are common to the next three algorithms, they will be omitted in their descriptions.

4.1.5 Pre-Copy Algorithm

The Pre-Copy algorithm[21] was invented by Theimer for V. It optimizes the transfer of the virtual address space to the new host by performing it in parallel with the continued execution of the migrating process at the old host. Assuming migration is accepted by the new host, the old host in the Pre-Copy algorithm ships all code, heap, and stack pages to the new host. If the total number of modified pages sent is greater than a predetermined limit, the process continues execution and newly modified pages are retransmitted. Otherwise, the process is suspended, and all other state information is transferred.

The algorithm reduces migration time by limiting the number of pages of the virtual address space transmitted during process suspension. However, the overall cost is increased if pages continued to be modified while the process is executing on the old host and need to be retransmitted. It may take several page shipments until the number of modified pages is less than the predetermined limit.

4.1.6 Demand Page Algorithm

The largest source of overhead and delay in previous algorithms is transmitting the virtual address space. Zayas implemented a Demand Paging process migration algorithm[23, 24] for Accent[15]. It uses lazy copying or copy on reference for pages similar to demand paging. The Demand Paging algorithm is similar to the Total Copy algorithm except that no virtual memory pages are transferred at migration time. All other state and file information is transferred but no virtual memory pages. When the process is restarted at the new machine, it will page fault immediately. The process then requests the desired page from the old host. The new process will likely page fault three times very early in its execution; one page fault for the current code, stack, and heap pages respectively. Future page faults are satisfied by requesting the pages from the old host.

This method ships less virtual memory data, and pages that are shipped are sent on an as needed basis. This results in shorter message freeze and process migration latency times and no wasted transfer of unused pages. The disadvantage is that the old host must maintain virtual address space information until the process completes. This may be a problem if a process migrates multiple times. By leaving address space information on the old host, a residual dependency exists, and the algorithm is not fault tolerant. Thus, the Demand Paging algorithm was an important breakthrough as it used operating system demand-fetch techniques to reduce transfer time, but it suffered from residual address space information at the old host which prevented it from being fault tolerant.

4.1.7 File Server Algorithm

The File Server algorithm developed by Douglis for Sprite[8, 14] was an important breakthrough in process migration. It added a third machine to the process migration mix, the file server. Previously, only the old host and new host were involved in process migration. By adding a file server, process migration can be done using the efficiency of demand paging without having a residual dependency on the old host.

The File Server algorithm is very similar to the Demand Paging algorithm. After receiving permission to migrate the process to the new host, the old host sends state information, link information, file descriptors, and address space information. The old host then tells the new host to start executing the process. Then instead of the old host maintaining the address space as in the Demand Paging algorithm, the old host concurrently flushes dirty pages and file cache blocks to the file server. Initially the old host may have to satisfy page requests, but eventually the file server will handle page faults generated by the migrated process when the address space is transferred off the old host.

Migration in Sprite is considerably more efficient than other implementations but still may take several seconds. Migration is also simplified because Sprite applications do not use messages and only use the file system for communication, which is not time-sensitive. Studies have shown the Sprite's file system communication is not as simple or efficient as Amoeba's message communication[16], so it is likely that future migration systems will have to handle message communication. Although the algorithm has the potential for long message freeze times, its performance is the best of the old systems and leaves no residual dependency.

4.1.8 Freeze Free Algorithm

The Freeze Free Algorithm[16] was proposed by Roush for his Ph.D. thesis. Its goal was to improve on current algorithms which fail to migrate processes faster than the average process execution time of 50ms. This section presents some of the motivations for the algorithm, the algorithm itself, and a critical analysis on the improvements it makes over current algorithms.

The major motivation for a new process migration algorithm is that latency times were too large in previous implementations to make process migration practical. With process execution time averaging 50ms, Roush reasoned that process migration should take no longer than half this amount, or 25ms. The Freeze Free algorithm can migrate a process in 14ms in its current implementation.

Achieving this speedup requires several improvements over existing process migration algorithms. The first is to eliminate the approval request message. Instead, the old host can assume the new host will accept the process migration and send the program counter and execution state to the new host. This saves time waiting for a reply which is a costly round trip message. Of course, the algorithm still must restart the migration process if the new host rejects the migration.

Transferring the address space is also a problem. Current algorithms either take an all or none approach to transferring the address space, even though a program needs at least a few pages to being execution. One alternative is to send the current code, stack, and heap pages because these pages are probably going to be needed right away. The current code page can be determined from the program counter, the current stack page can be determined from the stack pointer, and the current heap page can be estimated heuristically by examining instructions with load or store operations to registers. The other pages can be flushed to the file server concurrently with execution of the process at the new host.

Handling interprocess communication is difficult because a migrating process may receive messages. Currently all algorithms freeze messages because they are unable to separate process state from the state of the communication links. The problem is that the communication subsystem modifies process state on message receipt which may affect other data structures. Thus, message processing must be frozen when transferring process state. The solution to the problem is to remove the communication link state from the module containing the process state and vice versa. By allocating a separate memory region to hold the communication state and isolating the process and communication link from each other, message receipt can proceed in parallel with process migration. The communication subsystem buffers messages in the communication region and records relevant queue information but does not inform a migrating process of message receipt. The communication subsystem can then migrate separately which greatly reduces the message freeze time. Thus, message freeze time is no longer dependent on overall process migration time.

Messages may still be missed when migrating the communication link, so a flag is set to reject incoming messages. After migration, the old host rejects incoming messages by sending an error and the new location to the sender. The communication subsystem that sent the message updates the link information and retransmits. In rare cases, it may be possible that a retransmitted message arrives at the new host before the communication link, so messages are blocked at the new host until the communication link arrives. A migrating process can resume at the new host without waiting for the communication link to arrive and will block if it tries to access communications before it does arrive. This procedure effectively eliminates message freeze time.

Other improvements made by the Freeze Free algorithm include concurrent file cache management, initial structure allocation, and increased process modularity. The Freeze Free algorithm discards clean pages and flushes modified file cache pages in parallel with process execution at the new site. If the new process accesses a file page that is still at the old host, it is sent from the old host instead of the file server. This should happen infrequently as most file access is sequential, and thus fully written file blocks are unlikely to be written again. Also, data structures used for migration are allocated and partially initialized before migration to reduce latency time. To make process state extraction and insertion quicker, operating system data structures are organized into objects. This allows objects to be transparently transferred and edited quicker.

The Freeze Free algorithm is intended for a distributed system using message communication and supported by a distributed file server. All hosts are homogeneous and run the same operating system. The algorithm is as follows:

- Old host suspends the migrating process.
- Old host marshalls execution state and process control state and transmits it to the new host.
- New host initializes empty process object and responds with an accept or reject.
- Simultaneously and right after its initial transmission, the old host determines the current code, stack, and heap pages and transmits them.
- When the new host receives the first code, stack, and heap (optional) page it resumes the migrating process.
- Concurrently, the old host ships remaining stack pages, followed by communication links and file descriptor information.
- The old host then flushes dirty heap and file cache pages. (This can be done in parallel with transfers to new host if there is a separate communication link.)
- After all data is transferred off the old host, it sends a flush complete message to the new host.

The process migration latency time for the algorithm is the time from process suspension until the new host receives the first stack page. This time is constant regardless of address space size, number of open files, and number of dirty pages. Performance may be effected by the number of bytes truncated from the stack (above the top-of-stack pointer) and if the heap page is found.

Message handling is handled by the old host until the communication link is sent, and then the new host handles communication after the new host receives the link. Message receipt is never stopped but is only delayed while the communication link is in transit. Thus, message freeze time is effectively zero.

The address space is transferred by concurrently flushing pages to the file server and satisfying requests from the process. Thus, the process is minimally delayed by page requests, especially by initially providing it with its current code, stack, and heap pages.

In general, the Freeze Free algorithm appears to optimize the process migration process. This is demonstrated by a test implementation which is able to migrate a process in 14ms with a 4Kb page size and have a near zero message freeze time. This compares with a latency time of 41ms and a message freeze time of 21ms for the Demand Paging algorithm, and a latency time of 165ms and a message freeze time of 143ms for the File Server algorithm. The File Server algorithm is slower than the Demand Paging algorithm because it flushes the entire virtual memory before restarting process execution. In conclusion, the Freeze Free algorithm appears to be a significant optimization over current implementations.

4.2 Heterogeneous Process Migration

Homogeneous process migration allows good use of available network resources but is only applicable between machines with the same architecture and operating system. Many networks contain a variety of machines running different operating systems and provide other resources available on machines which have different architectures than the current machine where the process is executing. Using this computational power requires heterogeneous process migration.

Heterogeneous process migration is process migration across machine architectures and operating systems. Obviously, it is more complicated than the homogeneous case because it must consider machine and operating specific structures and features, as well as transmitting the same information as homogeneous process migration including process state, address space, and file and communication information. Heterogeneous process migration is especially applicable in the mobile environment where is highly likely that the mobile unit and the base support station will be different machine types. It would be desirable to migrate a process from the mobile unit to the base station and vice versa during computation. This could not be achieved by homogeneous migration in most cases.

There are 4 basic types of heterogeneous migration[18]:

- **Passive object -** only data is transferred and must be translated
- **Active object, migrate when inactive -** The process is migrated when it is not executing. The code exists at both sites, and only the data need be transferred and translated.
- **Active object, interpreted code -** The process is executing through an interpreter so only data and interpreter state need be transferred.
- **Active object, native code -** Both code and data need to be translated as they are compiled for a specific architecture.

One system implementing heterogeneous process migration is Emerald. Emerald is an object-oriented programming language based on the distributed object model Eden[1] which supports movement of active objects in a heterogeneous environment. Each object has a uniform view of the outside world and precompiled code for each target architecture. During migration data conversion is done on global data (data within the object) and local data (data arising from method execution). Emerald is a type-safe language which simplifies the migration. A **type-safe** language is an language where it is possible to uniquely determine the type of each data value within the program.

Another system called Tui[18] attempts to improve on Emerald by migrating programs written in type-safe C or Pascal, which are more popular languages than Emerald. C is not type-safe by its definition. For example, the union struct is not type-safe as it is hard to determine its contents at any given time. Other examples include the sizeof() call, void pointers, and pointer casting. Thus, the compiler must handle migration and inform the user if the program can be migrated. The current implementation of Tui migrates type-safe ANSI-C between machines running Solaris, SunOS, and Linux.

The migration process is relatively simple in Tui. First, the program is pre-compiled for each architecture. The program initially executes on the source machine. At migration time, the process is checkpointed, and the execution state is converted into an intermediate form which is written to a file and contains the variables, stack, and heap. The file is sent to the destination machine, and the data and state are reconstructed for the new machine from the intermediate form. The process is then restarted.

Obviously, there are many implementation details that must be considered. For example, the program counter may be different for different compiled versions of the program for different architectures. This prevents a program from being stopped at all places because some operations might take more or less instructions on a given machine. Tui resolves this by defining a set of preemption points in the code where migration is allowed. Temporary results from procedure calls are also tricky to handle, so every procedure call is logged by storing its address, temporary values, and parameters. Finally, reconstructing the stack and heap is difficult because the size of basic types may differ between architectures.

In terms of performance, encoding and scanning for the data used in the program takes most of the migration time. Rebuilding the data on the destination also takes a lot of time. All three of these operations grow linearly in the amount of data used by the program. Small programs can be migrated within 5 seconds which is very slow compared to homogeneous migration. Thus, the usefulness of heterogeneous process migration depends on the performance gains that can be realized by the migration. It is probably only useful for large processes and when the process is migrating to a more powerful machine.

4.3 Other forms of Process Migration

There are two other types of migration. Process migration can be performed using distributed shared virtual memory (DSVM) to aid in transferring process data and state. Also, object migration is possible in persistent objectbases.

Migrating a process is simpler using DSVM because the DSVM system provides a transparent way for migrating the address space and data. One example implementation[19] is an application-level migration mechanism implemented as a set of library calls over Parallel Virtual Machine (PVM). These calls perform freezing/unfreezing of communication, check load conditions, and handle migration problems. In this implementation, the programmer explicitly indicates the checkpoint data and asks the system to migrate the process. This has the advantage of being simple, portable, and transfers the minimal amount of data. Unfortunately, the migration is no longer transparent to the programmer who has to take increasing responsibility for its implementation. PVM provides data conversion and synchronization across heterogeneous platforms. The biggest win in using DSVM, such as in this implementation, is the transparent data conversion and transmission between source and destination machines performed by the DSVM manager. However, there may be problems with residual dependencies depending on how the DSVM manager transfers pages to the new destination.

Finally, object migration is possible in objectbase systems. In these systems, persistent objects are moved between machines depending on usage or placement statistics. The object may be migrated during execution, and many of the same problems in heterogeneous process migration arise during object migration. In these systems, migration is simplified by using the object-oriented techniques of encapsulation and well-defined user interfaces. This allows the system to easily determine the data and the execution state that needs to be translated and transferred.

5 Future Work

Process migration and load balancing algorithms have been around for quite a while. It is well-known what has to be transferred during migration, so process migration algorithms can only be improved by transmitting this data in the most efficient way. The Freeze Free algorithm appears to be a very general homogeneous process migration algorithm which optimizes transmission. It is highly unlikely that significant performance improvements on this algorithm exist.

However, determining when to migrate and measuring processor load still has important research applications. Deciding when to migrate represents the common problem in distributed computing of making decisions with incomplete information. Research into predictive measures of load would be useful. It may be possible to determine future load by examining past requirements and current process properties.

Heterogeneous process migration is a lot more interesting research area, but it remains to be proven that it is useful and worth the cost. A interesting benefit of research in this area is improving languages and compilers to provide better migration and heterogeneity support. Current languages like C are not easy to migrate and have no features to support migration or heterogeneity. By researching the requirements of such a language, it may be possible to define a language usable for today's widely distributed and heterogeneous networks.

Process migration over DSVM also has some interesting applications as DSVM systems become more available. Residual dependencies may be a problem if pages of a process are not immediately migrated by the DSVM system from the old machine during migration.

An important advanced application of process migration is in the utilization of networks of workstations (NOWs). Many organizations have massive computing power available when workstations are combined over a network. Unfortunately, current operating systems and programs do not take advantage of most of these resources. As an extension to process migration, process division would be useful. Process division would be similar to parallel processing except it is more transparent to the programmer. The goal would be the integration of a programming language/compiler, which allows the programmer to specify what parts of the process can be done in parallel or remotely, and a distributed operating system, which uses this specification and current processor loads to divide the process among various processors (maybe in a heterogeneous fashion) transparently at run-time for the user. This would result in increased machine utilization and some increased parallelism and performance for individual applications.

6 Conclusion

This paper is a survey of process migration mechanisms. Process migration involves transferring a running process between machines. In homogeneous process migration, this transfer is between machines of the same type, while heterogeneous process migration transfers processes between machines of different architectures and operating systems. There are many algorithms to implement process migration which attempt to minimize the process migration and message freeze times. Finally, although process migration is a well-developed area with limited areas of future research, research into compiler technology and programming languages which better support distribution, migration, and heterogeneity, when combined with a suitable process migration mechanism, may allow for better utilization of networks of workstations.

References

[1] G.T. Almes, A.P. Black, E.D. Lazowska, and J.D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, pages 43–58, January 1985.

[2] Yeshayahu Artsy and Raphael Finkel. Designing a process migration facility: The Charlotte experience. *Computer*, 22(9):47–56, September 1989.

[3] D. Butterfield and G. Popek. Network tasking in the LOCUS Distributed UNIX System. In *USENIX Summer 1984 Conference Proceedings*, 1984.

[4] D.R. Cherition. The V Kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[5] D.R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[6] Dasgupta, Chen, Menon, Pearson, Ananthanarayanan, Ramachandran, Ahamad, LeBlanc, Applebe, Bernebeu-Auban, Hutto, Khalidi, and Wilkenloh. The design and implementation of the Clouds distributed operating system. *Computer Systems*, 3(1):11–46, 1990.

[7] Dasgupta, LeBlanc, Ahamad, and Ramachandran. The Clouds distributed operating system. *IEEE Computer*, 24(11):34–44, November 1991.

[8] F. Douglis. Experience with process migration in Sprite. Technical report, University of California Berkeley, California, October 1989.

[9] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software - Practice And Experience*, 21(8):757–786, August 1991.

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

[11] Dan Freedman. Experience building a process migration subsystem for UNIX. In *Proceedings of the Winter USENIX 1991 Conference*, 1991.

[12] M. Litzkow, M. Livny, and M. Mutka. Condor – a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing*, 1988.

[13] Mark Nuttall. A brief survey of systems providing process or object migration facilities. *Operating Systems Review*, October 1994.

[14] J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[15] R.F. Rashid and G.G. Robertson. Accent: A communication oreinted network operating system kernel. In *Proceedings of the Ninth ACM Symposium on Operating System Principles*, 1981.

[16] Ellard Thomas Roush. The freeze free algorithm for process migration. Technical Report 1924, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1995.

[17] J.M. Smith. A survey of process migration mechanisms. Technical report, Columbia University, 1995.

[18] Peter Smith and Norman C. Hutchinson. Heterogeneous process migration: The Tui system. Technical Report TR-96-04, UBC Computer Science Department, Vancouver, B.C., February 1996.

[19] J. Song and H.K. Choo. Application-level load migration and its implementation on top of PVM. Technical report, National University of Singapore, 1995.

[20] A.S. Tanenbaum and S.J. Mullender. An overview of the Amoeba distributed operating system. *ACM Operating Systems Review*, 15(3), July 1981.

[21] M.M. Theimer, K.A. Lantz, and D.R. Cheriton. Preemptable remote execution facilities for the V-system. In *10th ACM Symposium on Operating Systems Principles*, 1985.

[22] Bruce J. Walker, Gerald J. Popek, Rober English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Ninth ACM Symposium on Operating Systems Principles*, 1983.

[23] E. Zayas. Attacking the process migration bottleneck. In *11th ACM Symposium on Operating Systems Principles*, 1987.

[24] E. Zayas. *The Use of Copy-on-Reference in a Process Migration System*. PhD thesis, Carnegie Mellon University, 1987.