Process Migration in FROST

MICHAEL GLIBSTRUP - LARS KRINGELBACH

DEPARTMENT OF COMPUTER SCIENCE
AALBORG UNIVERSITY

TITLE: Process Migration in FROST
THEME: Distributed Systems
SEMESTER: F10S
PROJECT TERM: February 2002 - June 2002

Abstract

FROST is a distributed heterogeneous calculation platform first described in [GK02]. It provides an API that allows a user to create assignments which are distributed and calculated on a number of computers.

In this project we have chosen to improve the dynamic load balancing scheme of the FROST system by implementing process migration. We do this in order to improve the performance of the system as processes can be moved from nodes that are heavily loaded onto nodes that are less loaded during execution.

We have analyzed the problem of process migration and have decided on a solution using checkpoints as a means of performing process migration. The chosen solution has been designed and parts of the design has been implemented. During the design phase we have been very aware that the performance aspects of FROST should not become worse after the introduction of process migration as it would remove the reason for using the feature.

We have tested the system with regard to overhead, performance and the behavior of the migration policies. The tests show that the overhead with regard to the process migration features is fairly small compared to what they offer. In addition the performance tests and the policy tests showed that the system performs well after the introduction of process migration. We do conclude, however, that in order to obtain even better performance it should be considered to introduce migration points as a tool as well.

AUTHORS:
Michael Glibstrup
michaelg@cs.auc.dk
Lars Kringelbach
lars@cs.auc.dk

SUPERVISOR:
Josva Kleist
kleist@cs.auc.dk

NUMBER PRINTED: 6
NUMBER OF PAGES: 116
FINISHED: June $7th$, 2002

Dansk Resumé

Indenfor feltet omhandlende langtidsberegninger ved brug af ikke-dedikerede maskiner er det kendt, at mulighed for dynamisk redistribution af belastning kan hjælpe med til at forbedre ydeevnen i systemet. Dette skyldes inflydelsen fra brugerprocesser, som ikke kan forudsiges i den indledende tildeling af opgaver til maskiner.

FROST er en distribueret heterogen beregningsplatform, som er beskrevet i [GK02]. Den er tiltænkt at køre i et ikke-dedikeret miljø, hvor den bruger de ledige CPU cykler på almindelige arbejdsstationer til langtidsberegninger. FROST stiller et API til rådighed, som tillader en bruger at skabe opgaver, der bliver distribueret og beregnet på et antal computere, hvilket muliggør udnyttelse af ubrugte CPU cykler i et lokalnetværk.

I dette projekt har vi valgt at forbedre FROST systemets dynamiske load-balancering ved at implementere procesmigrering. Vi gør dette for at forbedre systemets ydeevne, da processer kan flyttes fra knuder som er hårdt belastede, til knuder som er mindre belastede under udførelsen af dem.

Vi har analyseret procesmigreringsproblemet og har valgt en løsning som bruger checkpoints som et middel til at udføre procesmigrering. Den valgte løsning er designet og dele af den implementeret. I løbet af designfasen har vi været meget opmærksomme på, at ydeevnen i FROST systemet ikke blev forværret efter introduktionen af procesmigrering, da dette ville fjerne grunden til at bruge denne egenskab. Derudover er måder, til at gøre FROST skalerbart på, blevet overvejet.

Vi har testet systemet med hensyn til overhead, ydeevne og opførslen af migreringspolitikkerne. Testene viser at overheadet med hensyn til checkpoints og procesmigrering er ganske lille sammenlignet med, hvad der tilbydes af procesmigreringsegenskaben. Vi konkluderer dog, at for at opnå endnu bedre ydeevne burde det overvejes ligeledes at introducere migreringspunkter som et værktøj.

Preface

This report is a master's thesis based on work performed on our 10th semester in computer science, distributed systems. The purpose of the report is to communicate the thoughts and used approaches in considering, analyzing, designing, and implementing process migration into a distributed heterogeneous calculation platform, the FROST system. The work done in this master's thesis builds on work presented in [GK02].

The report is directed towards people with interest in distributed calculation platforms, peer-to-peer computing, and process migration in heterogeneous environments.

In part I we will survey the area of process migration and checkpointing and from this determine the most optimal solution for FROST. We will also look upon the rules for controlling the migration and checkpointing. Part II contains performance considerations and the design of the elements necessary for implementing process migration in FROST. In part III we look upon the details regarding the implementation of process migration in the FROST system. Finally in part IV we perform tests of FROST system after process migration has been implemented, we consider how to make the FROST system scale to a larger platform, and we present a conclusion on the results. A word list can be found in appendix A describing some of the FROST terminology defined in [GK02]. Test results are located in appendix B.

The bibliography is situated in the back of the report. References to the bibliography are as follows: [GK02], which refers to the paper *"FROST - A Distributed Heterogeneous Calculation Platform"* written by Micheal Glibstrup and Lars Kringelbach in 2002. References to figures are made as follows: "figure x.y", where x is the chapter and y is a consecutive numbering within each chapter. References to tables are similar to the references to figures.

Some typography is used in the report in order to clarify meaning. Class names are written in Sans Serif. Algorithms in part I are described in pseudo-code using the *Algorithmic* environment. References to the algorithms in the report are as follows: "algorithm 1". In the algorithms, setting `var` before a variable in a method decleration means that the variable is passed as a reference to the method. FROST is developed in C++, and in part III, Implementation, examples are therefore given in a C++-style language.

Throughout the report we use the words computer, machine, workstation, and node interchangeably.

Michael Glibstrup Lars Kringelbach

I'm a great person for utilizing waste power

- Robert Lee Frost

Contents

CHAPTER 1

Introduction

The FROST system was designed and implemented in [GK02] as part of our 9th semester project work, with systems such as SETI@home and Distributed.net in mind. The aim was to develop an API that could be used to easily develop SETI@home-like applications, which are automatically distributed to machines on the Internet[1] by the built-in runtime system. The system is designed to be used Internet-wide using non-dedicated machines. A system with this setup sets some demands to efficiency and durability. As the system is designed to run on non-dedicated machines it requires that the application is fault-tolerant as machines can break down. Because it is to perform Internet-wide the system demands a high calculation to communication ratio in order to achieve optimal performance. In the following we will give a further description of the FROST system and a characterization of the applications that are suitable to the system. Finally we will introduce the work that is contained in this report.

1.1 The FROST System

The FROST system is a distributed calculation platform that exploits unused CPU-cycles in a network of non-dedicated workstations. Non-dedicated means that the machines are normally used by local users and these users must be given highest priority when they need the machine. As FROST therefore has low priority on the machines, more machines are necessary than in a system that uses dedicated machines in order to obtain the same amount of computing power. To make most machines available to the system, heterogeneity is considered to be an important factor, and therefore it has been made easy to develop portable applications to the FROST system by letting FROST take care of non-portable issues such as byte-order conversions, distribution of binary files, etc.

The system is intended to be used widely on the Internet, providing lots of users with access to surplus computing power. For this reason, the system is based on the peer-to-peer paradigm where all machines performs on an equal basis. This is implemented so that in order to utilize computing power on other machines, a computer must provide computing power in return.

Due to the use of non-dedicated machines, we cannot count on CPU-cycles being available for each application at all times. Some calculations, or part of a calculation, can be postponed or slowed down if the amount of available CPU-cycles is small. For this reason we have dedicated FROST to performing calculations and therefore it cannot be used for applications which require any interaction from users or other external devices.

When developing applications for FROST the user only has to concentrate on his application domain. He needs to provide an algorithm for splitting the problem into several pieces, or *work units*, that can be processed independently and an algorithm for combining the results when they have been processed. Finally he must specify the algorithm that performs the calculations. Everything regarding the distribution of work units between the available machines, including all network communication, is performed by FROST. Dynamic load balancing is performed in order to have applications finish in the shortest possible time.

The interprocess communication capabilities available in FROST introduces some limitations to the development of applications to FROST. In the following section we will describe interprocess communication in FROST and afterwards the applications that are suitable for the FROST system are characterized.

[1] The applications are only distributed to other machines that have the FROST system installed.

1.1.1 Communication in FROST

FROST was designed with the possibility for limited interprocess communication. The communication consists in passing on a sub-result to another calculating node. This node is then able to use the previously calculated result in the further calculations. All communication must be known in advance, that is, the dependencies between result and calculations must be known. These dependencies must be specified using the graph structure included in the FROST system. FROST will then handle all interprocess communication from the dependencies specified in the graph.

Because communication must be specified beforehand there are some limitations to the problems that can be solved using FROST. Hence, applications that distributes a data structure across several machines due to lack of memory on a single machine, will often not be possible to implement in FROST. If the data structure is accessed randomly, the communication cannot be specified in advance which is necessary.

An example of problems that can take advantage of the communication available in FROST is the class of dynamic programming problems. In this class of problems earlier results are used in the further calculations in a predictable way. An example is the parenthesizing of matrix-chain multiplication in order to find the multiplication order with the least amount of multiplications. For these calculations the tables shown in figure 1.1 are used.

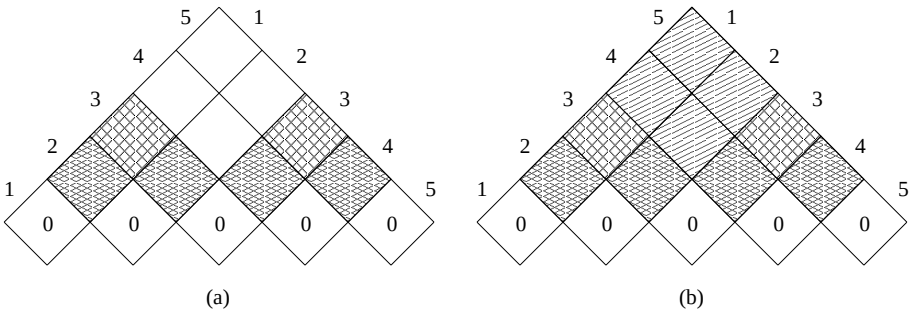


Figure 1.1: Calculation of matrix-chain-order. a) First the dark shaded areas are calculated in parallel. Afterwards the two light shaded areas are calculated in parallel using the results from the dark shaded areas. b) Finally the rest of the table is calculated using the previously calculated results.

The dark shaded areas in figure 1.1 (a) can be calculated in parallel. The results from these calculations must then be transferred two by two to the machines calculating the light shaded areas, e.g. the results in place (2,1) and (3,2) is transferred to the machine calculating the result for (3,1). Finally all the shaded areas in figure 1.1 (a) are used to calculate the last part of the table as shown in figure 1.1 (b).

Even though this kind of problems is suited for the interprocess communication primitives available in FROST, they are not especially suited to be solved with FROST. There are several reasons for this:

Imbalanced distribution: Using the procedure described above will make the work units larger for each level of calculations. There is however a solution to this. Instead of calculating the striped area in figure 1.1 (b) at once, it can be split across several machines. This can be done by first calculating (4,2) using (3,2) and (4,3), and afterwards (4,1) using the entire underlying triangle and so forth. This will solve the imbalance of calculation sizes, but it will require more transferral of data between machines.

Memory usage: The most optimal problems to FROST uses less memory to process a work unit on a machine than is used when processing the entire problem on a single machine. We cannot expect that the workstations have the same amount of memory as a supercomputer and therefore this is an important property, especially if the problem is very memory consuming. The above example does not have this property, since when calculating a part of the table, the entire table

in a triangle below must be available. When calculating the last part, the entire table must be available as if all of the table was calculated on the same machine. This applies for dynamic programming problems in general and will be a problem for large problems.

Communication: The calculation to communication ratio for the above problem is not optimal. In order to achieve maximum parallelization the amount of sub-results that needs to be sent increases. The size of the sub-results increases when moving up the table, and therefore a large amount of communication is necessary.

Parallelization: An entry in the table can not be calculated until all entries in the entire triangle below have been calculated. This limits the amount of parallelization possible, which can give poor performance in FROST, as a low bandwidth, high latency network is to be used.

The above issues apply to the entire class of dynamic programming problems and this class of problems is therefore not suited to be solved with FROST. It is important to mention that the problems are well suited to be solved in parallel if a shared memory machine is used, as can be seen in the literature. Some examples are Hardwick et al. [HOS99] and Andonov et al. [ARQ93].

We have not been able to find any problems that can be solved efficiently with FROST *and* uses the possibilities for communication. FROST has been designed to be used on a network of *non-dedicated* workstations connected with a *low-bandwidth* network[2], and these conditions sets some limitations to the problems that are suitable to be solved. For this reason we have chosen not to support the limited communication that was designed in [GK02], and instead optimize the system for problems which do not have any interprocess communication.

In the following section we will characterize problems that are suitable to be solved with FROST by first describing the systems that we had in mind when FROST was designed.

1.1.2 Applications in FROST

In [GK02] we designed FROST with projects such as SETI@home, Distributed.net, and THINK in mind. These projects solve problems that are all optimized to be solved using non-dedicated workstations connected to a low-bandwidth network. In the following we will characterize these projects and the problems they solve. The problems all have common features. They can be split in a number of work units where each work unit can be processed completely independent of another, hence there is no need for interprocess communication between the different machines used to solve the problem. They also have a very high calculation to communication ratio which makes them extremely suitable for a low-bandwidth network connection. The following is a description of some of the projects:

SETI@home: SETI@home[3] is a project set on finding radio signals from alien civilizations. To obtain the data, the large radio telescope at Arecibo is used. This telescope scans the sky, recording data in a certain frequency range. When enough data has been gathered, the data is split up into 10 kHz work units of about 100 seconds in length. These work units are then sent to computers over the Internet for further processing[4].

A work unit contains data to be calculated upon for a single machine. These work units can be calculated independently of each other which makes communication between machines unnecessary.In the SETI@home project they contain 340kb of data and they should take between 10 and 50 hours to complete on an average home computer. Currently the average CPU-time per work unit is 17 hours and 14 minutes.

Distributed.net: Distributed.net consists of several projects which are mathematical in nature. The projects are mainly on the effort of cracking encryption ciphers of different encryption schemes,

[2] In [GK02] the system was designed to a 10-100Mbit network. If the system is to be used Internet-wide we cannot expect the machines to be connected with a 100Mbit network, but rather a 0.5-10Mbit connection or lower.

[3] `http://setiathome.berkeley.edu`

[4] For further information on the Seti@home project and the algorithms that are run on the data see `http://setiathome.berkeley.edu/about_seti/about_seti_at_home_1.html`

but a project of finding the most optimal Golomb rulers[5] is also in progress[6].

The work units in the Distributed.net projects are of very different size. In the RC5 challenge the user can determine how much work he wants to retrieve at a time dependent on how often he connects to the Internet and how fast his computer is.

THINK: The THINK[7] project covers a wider selection of problems that needs to be solved. Examples are researching Anthrax, Cancer, understanding the human genome and other medical research problems as well as web performance testing. Common to the medical problems are that they utilize the same algorithms for performing calculations on a dataset over and over again. For example in the cancer research program, computers test a number of molecules in order to find a drug that could cure cancer. The project regarding the human genome utilizes Hidden Markov Models to find genetic sequences that can be related to each other in some way.

In the testing of molecules, a work unit contains approximately 10Kb of data and the average CPU-time per unit is currently 7.5 hours. The CPU-time required varies from 4 hours to several days only due to differences in the data that is calculated upon.

All of these problems lie within the class of problems called *embarrassingly parallel*, as described in [FWM94]. Embarrassingly parallel problems have a simple spatial structure which leads to clear parallelization, and there is no synchronization involved and hence no interprocess communication is necessary. The only communication needed is to set up the problem and accumulate the results. Due to the lack of synchronization between machines it is often possible to archive a linear speedup, especially if the problem has a high calculation to communication ratio. Another example of embarrassingly parallel problems is Monte Carlo simulations [Cod93].

This kind of independent or "job level" parallelism is common to many kinds of simulations in science and engineering. Sometimes it is done by performing similar calculations on each machine but with varying parameters if possible. This approach is ideally suited to very coarse-grained parallel machines, and especially for distributed computing using networks of workstations [Cod93].

1.2 Further Work

In [GK02] we have noted some of the further work necessary in order to achieve a more optimal system. These are summarized in the following.

Communication between slaves: The possibility of communication between slaves was never fully implemented and we have chosen not to support this feature in future versions of the system.

Load balancing: FROST uses a dynamic load balancing scheme where the initial assignment of work units to nodes can be changed during runtime as needed. In [GK02], we say that if there is communication between slaves an intelligent initial assignment of tasks to machines is of great importance in order to minimize the expensive interprocess communication. As we do not support communication between slaves, we believe that our dynamic load balancing scheme performs reasonably compared to an intelligent initial assignment as the load can vary a great deal during calculations. This is especially true for small tasks as this will minimize the time needed on a heavily loaded machine. It is, however, more convenient with larger tasks[8] as it will minimize the amount of communication between slaves and the master and hence the administration load on the master. Larger tasks will therefore make it possible to achieve better performance. This contrast requires new means for load balancing in order to achieve optimal performance.

Fault-tolerance: Fault-tolerance has not been implemented and the problems due to failing machines can be minimized by having smaller tasks. As mentioned above larger tasks can provide better performance and is more suitable to the FROST system. This will however make the

[5] Golomb Rulers are described in detail at `http://members.aol.com/golomb20/intro.htm`
[6] For more information on the distributed.net projects see `http://www.distributed.net`
[7] `http://www.ud.com`
[8] By large tasks is meant that a single task takes long time to compute. *A long time* is in this context several hours.

need for fault-tolerance more necessary, as a larger part of a calculation can be lost if a machine fails.

Security: Security was given a low priority in the design phase of FROST as it is not important in order to show the applicability of the system. It is, however, very important in a final version of the system both to secure the privacy of the data being calculated upon and to have secure execution of foreign code on slave machines.

Portability: The portability issue was also given a low priority in the design phase for the same reason as with security. Some means for increasing the portability have, however, been implemented, such as byte order conversions when transferring data over the network. If the calculation code is made portable it should be fairly easy to port the system to other UNIX-based platforms.

In the following section we will elaborate on the issues described above in order to determine the work we will proceed with in this project.

1.3 Problem Statement

We still see the security issue as important in a final version of the system, but as we are still in the initial phases of the construction of a generic distributed calculation platform, we will give it a low priority.

This project will consider the improvement of issues regarding load balancing and fault-tolerance, as these have direct influence on the performance of the system. By implementing fault-tolerance techniques it is possible to distribute larger tasks which will minimize the amount of communication needed with the master and hence decrease the bottleneck the master imposes. With larger tasks the need for a more intelligent load balancing scheme increases

In order to make it an advantage to have larger tasks some requirements are set to the load balancing scheme. We cannot predict the load on non-dedicated machines as local users can decide to use the machine at any time. The FROST system was designed in [GK02] to run with a low priority to ensure that user processes always will have precedence over FROST tasks. When a user loads his machine heavily the FROST tasks will not gain any CPU-time. When a single task has a long running time, the implemented dynamic load balancing scheme will not suffice, since balancing can only happen between tasks. Instead of the dynamic load balancing scheme already implemented in FROST the use of process migration should be considered in order to achieve better performance. According to Smith and Hutchinson [SH98] it is an advantage to use process migration when a system mostly executes longer running tasks. Process migration works by moving a calculating process to another machine at runtime. By supporting process migration we can not only balance the load when a new task is sent to a slave, but also change the assignment of tasks to nodes during execution.

When most of the tasks in the system are long running it is necessary for the system to be fault-tolerant. This would protect the tasks from e.g. a system break down or a machine shutting down during a calculation[9]. In order to do this it must be possible to revert the process to a known state before the failure. This means that there must be some kind of checkpointing where the state of the process is saved to a fail-tolerant media.

Process migration is basically capturing the state of a process to be transferred and transferring it to another machine where it is continued. Checkpointing is a method of capturing the state of a process and saving it to disk e.g. for fault tolerance. Thus, a checkpoint that is transferred to another machine instead of saving it to disk is a way of performing process migration.

We have chosen to concentrate on implementing process migration in the FROST system as we believe that this issue is very important to the performance of the system.

[9] This kind of failure is known as fail-silent or fail-stop in the literature [Tan95]

PART I

Analysis

This part presents an introduction to the area of process migration and a discussion of how to introduce process migration into the FROST system. In chapter 2 we survey the area and look at different ways of performing process migration. In chapter 3 we look at the different possibilities presented in chapter 2 and discuss them in relation to FROST.

CHAPTER 2

Process Migration

In the following we will survey the area of process migration which allows tasks in a system to move from one computer to another as need arises. We will consider some of the advantages and disadvantages of process migration. We will also reflect on some of the ways that other systems perform process migration. Later we shall see that fault-tolerance can be introduced through the use of checkpoints.

2.1 Motivation

The need for process migration can be based on several things. In order to state what motivates the use of process migration in a distributed system we look at some examples.

- Another processor emerges which offer better performance than the present one.
- The current processor is about to shut down.
- The desire to minimize the amount of communication between processes on different computers.
- Memory demands that are not fulfilled by the current computer.
- Special architectural characteristics offered by some processors.

Process migration allows a task to gain access to these features by moving it from one processor to another.

An important aspect to keep in mind when considering whether process migration is an option for a distributed application is whether it offers a reasonable performance gain. It is obvious that if process migration raises the execution time of a process without offering anything else in return it will not be used. Preferably it should increase the performance of the system so that processes will finish faster when using process migration than when it is not used. The increase in performance can be gained by moving a process closer to another process with which it is communicating heavily, or moving a process from a heavily loaded computer onto a computer which is not loaded. If process migration cannot provide an increase in execution time it has to offer something else in order to justify including it in an application. This can for example be that a process will always be executing if needed except if the host computer breaks down. If process migration is coupled with checkpointing even a computer break down can be survived as the process can be recovered later.

When considering process migration it is necessary to take the cost of migration into account - how much is moved, how long will it take and how often is it done. In other words it is not enough just to move a process in order to secure a performance gain or a certain amount of memory, it is also necessary to consider the implications of the migration. If the implication of introducing process migration is too great it may be necessary to reconsider the introduction of it. We will consider the subject of performance further in a later section.

2.2 Performing Process Migration

Process migration can be performed in a number of ways, all depending on the architecture and approach used. In the following we will look upon some of the ways in which process migration can be performed.

2.2.1 The State of a Process

When a migration of a process is to be performed, it is necessary to transfer, not only the code of the process, but also the state of the process. The state consists of the process variables, pointers to other objects, file handlers, communication with other computers, call stack, and so on. All this has to be transfered to the system on which the process is to continue its execution. The basic way of performing this transfer is to stop the process that is to be migrated, extract the state of the process, inform the computers with which it is communicating, and then pass the state information along to the receiving computer. If checkpoints are used it is more complicated to handle communication between computers. This will be discussed further in a later section. Exactly how the extraction is done may differ from system to system but there are two basic ways of performing state extraction - direct and indirect extraction.

Direct Extraction

The direct approach of extracting the state of a process covers systems that, in some way, accesses the stack and heap of a process. A system like Condor [LTBL97] use this approach when migrating processes. Condor produces a file in which all information about registers, data and stack, pending signals and open files are displayed. This is then transferred to another computer where the process is restarted and its state restored. All system calls are performed through a shadow process which remains on the sending computer. The advantage of the Condor approach is that migration can be performed transparently from the user's point of view. The disadvantage is that when transferring the stack and data areas of a process directly, it limits the heterogeneity that a system is capable of providing, unless massive translations are made of the state information of processes at migration time, or a generic stack and heap is maintained in parallel to the normal ones.

Indirect Extraction

Opposed to the direct way of performing process migration is the indirect way. This way of performing process migration allows a more user-oriented approach as the preliminary work of the migration is performed by a preprocessor or the user himself.

An example of the indirect approach is the Dome-environment [ÁBL+95]. Here the user must conform to a predefined API which creates a Dome-environment. This environment then performs the necessary operations in the migration process. Another example is MpPVM [CS96]. MpPVM employs a preprocessor which translates PVM[1] source code into MpPVM source code by dividing it into subsequences and inserting migration points between these subsequences. It also performs data analysis in order to determine what data is needed at a specific migration point and thus must be included in the migration process.

The indirect approach consists of modifying the code of the process, so that variables and pointers are saved in a way that makes it possible to restore them after migration. This can be done using a table containing all variables and their types. Then at migration time this table can be transfered to another computer where the variables are reloaded.

The advantage of the indirect approach is that migration is performed at a much higher level and therefore it can be much more flexible when it comes to heterogeneity.

Seligman and Beguelin [SB94] gives an example on this in relation to the Dome system. They state that using a low level, or direct, method for checkpointing produces larger checkpoints than does their high level method. This is because a low level method tends to provide only page-level data granularity and tends to save a large quantity of other information, such as the full stack contents. A high level method on the other hand, enables a more selective approach to choosing which variables to include in a checkpoint. They give an example where their high level Dome approach gives a 10kb checkpoint file compared to a low level approach giving a 3.3Mb checkpoint file. Thus we can see that the closer the migration process is placed to the applications being migrated, the better is the prices that can be achieved.

[1] Parallel Virtual Machine

The disadvantage is that being such a high level approach it might jeopardize the transparency from the user's point of view. Migration demands an effort from the user in some cases and can thus be a problem rather than an advantage. This will be considered next.

2.2.2 Kernel or User Space

A certain level of transparency is important in a migratory system in order to make the system as easy to use as possible. But how do we achieve this transparency? In the previous section we considered the direct and indirect approach to process migration. Another possibility is to consider where the migration process is placed within an environment. There are two possibilities - kernel and user space.

If the migration procedure is performed in the kernel of an operating system it is easier for a system to migrate processes. As the kernel has direct knowledge of all processes, it can perform process migration independently of the user space code by moving the runtime stack, data areas, code and process registers directly onto another computer. This can be an advantage as the user or the processes in user space need not have any knowledge of the migration features that run in the kernel. In addition migration control can be coupled with the ordinary process scheduler in the kernel. A disadvantage is that if process migration is to be introduced into an existing kernel it requires extensive modifications to the kernel [MDP+00], and thus redistribution of the kernel. The modifications to an existing kernel could be avoided by creating a new kernel which already contains support for process migration but such a kernel would also require a distribution phase and would probably have to be incorporated into an existing system. Another disadvantage is that when performing process migration in kernel space it is much harder to migrate processes in a heterogeneous environment. Due to the possibility of having different architectures in a heterogeneous system it is no longer possible for the kernel just to move a process. Before migrating a process the state and code of the process has to be changed into a format which is understood by another architecture. An indirect approach cannot be used in kernel space, as the kernel has no knowledge of variable names and types.

Another approach is to place the migration process in user space. Milojicic et al. [MDP+00] divides the user space approach into two sub-categories - user-level and application specific migration. Both these levels generally suffer from lack of transparency, reduced performance and higher migration costs. They have the advantage though, that the closer the implementation of migration can get to the applications which it is to migrate, the more knowledge it usually has about them. This knowledge can be used in the derivation of better migration policies and hence, better overall performance.

Whether to use a kernel or a user space approach depends heavily on individual needs. If a migration tool that can migrate all kinds of processes is needed or if a very high amount of transparency is required, a kernel-space approach may be preferable. If on the other hand a more specialized approach is needed, where only processes from a single application needs to be migrated or if the migrations need to be strictly controlled or tuned to specific needs, a user-space approach may be chosen. When considering kernel or user space migration it should also be noted that the closer to the kernel the migration process is placed within the system, the harder it will be to achieve heterogeneity [MDP+00].

2.3 Preprocessor

In a user level approach to process migration, a preprocessor might be useful in order to insert migration primitives into the code of the user. This would enhance the system transparency in relation to the user. System transparency in relation to the user is a highly desired feature when considering process migration due to the high complexity of the migration feature.

Chanchio and Sun [CS96] uses a preprocessor approach in the MpPVM system. Here the PVM primitives in the user code is first translated into MpPVM primitives, then the code is analyzed and migration points are inserted. After this, data analysis is performed in order to locate *necessary* variables. Necessary variables are variables which are initialized before the migration point and which will be used after the migration point. Thus, if a variable is not used after a migration point it will not be transfered during the migration which means that only the necessary data is sent. After

the migration points and their data have been established, macros used for migrating and reinstating a process are inserted into the code.

A preprocessor changes the code created by a user. This can be an advantage as it is possible for a user to review and change the code created by the preprocessor before it is compiled into an executable. This allows the user to maintain the general view of the code and if needed, optimize the code generated by the preprocessor before it is compiled. If for example the user locates a place in the machine generated code where the preprocessor has inserted unnecessary migration points he is able to remedy this before the code is compiled.

The problem of using a preprocessor is that it can be quite difficult to read and understand machine generated code, and thus optimize or debug it[2]. In order to do so the user needs detailed knowledge on the workings of the preprocessor. This includes the API of the code introduced by the preprocessor, and where in the original code it might be included. If a user does not have this knowledge it is hard for him to perform debugging or optimization when the code has been processed by the preprocessor. This is because of the lack of correspondence between the original code and the code compiled by the compiler. Line numbers will be skewed and a translation scheme between original and modified line numbers would be a good idea for debugging purposes.

A preprocessor approach can be used without regard to the underlying architecture. This means that no matter if a homogeneous or heterogeneous architecture lies beneath the user code, a preprocessor can be used as it is only modifying the users code with predefined primitives.

2.4 Checkpointing

Checkpointing is performed by capturing the state of a process in such a way that it is possible to rollback the process and restart it at the point where the checkpoint was made. In order to do this, the checkpoint must include the entire state of the process such as the runtime stack and data areas, processor registers and so on.

Checkpointing is usually used to increase fault-tolerance as in Dome [ÁBL+95] and Condor [LTBL97] by saving the checkpoint to stable storage. In case of machine failure, the process can be restarted using the last checkpoint instead of starting over. The functions carried out when performing the checkpoint are very similar to those of process migration where the state of the process is transferred to another machine instead of being saved to stable storage. This feature makes it possible to use the checkpointing mechanism for process migration, just by transferring the checkpoint data to another machine and load it there in order to continue execution.

In spite of the similar approaches to process migration when using checkpoints or migration points, the methods have some differences. When using checkpointing, the checkpoint is made at each point specified in the code, where process migration using migration points, as in MpPVM, only capture the state when the actual migration takes place. This induces some runtime overhead in the checkpointing method as the state is captured more often. This is, however, necessary to obtain the fault-tolerance capabilities. Furthermore it gives the possibilities to migrate without waiting for the next migration point, the last checkpoint can be sent exactly when migration is wanted.

When using checkpointing, interaction between processes introduces a serious problem. If a process is restarted from a checkpoint, it can be necessary to rollback other processes in the system with which it has been communicating. This is a problem as there is a potential risk that all processes in the system needs to be rolled back to where they were when the checkpoint was performed, thus rendering the performed calculations useless. If all the communicating processes acts completely deterministic there is a solution to the problem. All the communication messages received between the checkpoint and the point where the process was stopped can be buffered and re-sent to the recovering process and thus no processes needs to be rolled back.

When using checkpointing for process migration, it is also necessary to consider the heterogeneity of machines in the system just as when regular process migration is used.

[2] The code inserted by the preprocessor must be error free so that only the code of the user needs to be debugged.

2.5 Process Migration Policies

In the above sections we have considered the more technical aspects of process migration. But in order to ensure that process migration becomes an advantage and not a nuisance it is necessary to consider when migration should be performed and where a process should migrate to.

Because of the fact that the load in a system is likely to change in a distributed system consisting of non-dedicated workstations, it is natural to consider process migration with regard to adaptive load sharing in these systems. But we cannot allow any process on one computer to migrate to another computer whenever it sees fit to do so. Therefore it is necessary with a set of rules which declare when and where a process should be allowed to migrate. These rules are known as the *transfer policy* and the *location policy* [C+01].

The transfer policy chooses the process that is to be migrated from one computer to another. The policy regards the load on the computer on which the process is situated, the size of the process being transfered, the time a process has run relative to the time it will run, and other relevant considerations.

When a process is chosen for migration the location policy is used to determine which computer in the system the process should be transferred to. When choosing a new computer for a process the location policy has to take different conditions into account. Examples are the relative load of the computers, differences in machine architectures, and any special resources they may possess.

Finally in addition to the other two policies, a third policy could be considered. This is the *information policy* which handles the gathering of information about the load in the system. This policy can be either centralized or decentralized. In the first case, a load manager is placed at one point in the network, collecting load information about all the other machines in the network. In the second case each node maintains a database of information and they exchange information with one another directly in order to keep the database up to date.

The three policies presented above are essential to the system in which process migration is considered as a tool and we will therefore consider them further in the analysis.

CHAPTER 3

Analysis

When considering process migration in relation to FROST, the issues introduced in the process migration survey in chapter 2 must be considered. These issues regard both technical and algorithmic properties when designing and implementing process migration in FROST. In the following sections we will analyze these issues in the context of FROST in order to set some demands to the further design and implementation.

3.1 Kernel or User Space

From section 2.2.2 we can identify three main issues considering the location of the process migration feature in relation to the FROST system, transparency, heterogeneity and portability.

The kernel space solution provides a great deal of transparency as it is most often implemented with a direct approach for extracting the state of a process. Normally the kernel has no knowledge of the variables in a process and therefore it can only access the process image as a whole. This issue makes heterogeneity very difficult to implement without providing the kernel with extra information.

If the process migration should be implemented in the kernel, it would make the FROST system dependent on the kernel it is running on. Hence, FROST would only be able to run on systems using the modified kernel. This would limit the portability of the FROST system a great deal, and probably make it impossible to port to Windows systems. By placing the migration feature in the kernel we would also have to consider the possibility that processes other than those from FROST could be migrated, which is not what we are interested in.

Opposite of placing process migration in kernel space, is incorporating process migration directly into the application in question. As stated by Milojicic et al. [MDP+00] the closer the implementation of process migration is placed on the applications which is to be migrated, the more knowledge it can obtain about the application. This knowledge then leads to better overall performance when it comes to migrating processes. Furthermore the extra knowledge is a very important factor when performing process migration in a heterogeneous environment.

As the heterogeneity and portability issues are very important to the FROST system, implementing the process migration feature in kernel space is not an option. By placing the migration feature in the FROST system itself we have complete control of the migration elements and we have direct access to the processes that is to be migrated. In order to obtain heterogeneity, we need to use an indirect approach for extracting the state of the process. In this way we have access to the variables and their types, which is necessary in order to transfer them across different machine architectures.

Placing process migration in user space introduces some problems. As stated earlier in section 2.2.1 process migration can be performed more or less transparently depending on the approach taken. By using an indirect approach it is necessary to analyze and modify the code of the process that is to be migrated. This fact reduces the transparency, but is necessary to fulfill the heterogeneity and portability issues. The transparency problem can, however, be solved by using a preprocessor to modify the source code of the process.

3.2 Preprocessor

The design of the FROST system is directed towards easing the implementation of distributed computing assignments. With this in mind it is of great necessity to consider the amount of transparency needed when introducing process migration into the system. If the user has to implement migration features into his source code, in addition to splitting his assignment, we have to contemplate the pos-

sibility that process migration may not be used. This is due to the fact that process migration is an advanced feature and any errors in the migration process may ruin the results of the calculation, if it completes at all. Therefore a high level of transparency is needed in the process migration procedure in order to aid the users as much as possible.

As stated in section 3.1 a preprocessor can be used to provide the required transparency, as the chosen placement of the migration feature requires modifications to the user code. The use of a preprocessor does however introduce another advantage. We are interested in providing the user with an approach which enables him to consider the most optimal placement of checkpoints, but which performs the actual data analysis and process migration without any involvement from the user. The reason why the user has to state the point where a checkpoint is to be performed, is that he is the one which holds the most knowledge of the current problem, and thus can place the checkpoints so that optimal performance is achieved. This approach is not completely transparent, but it is sufficiently transparent to allow a user to use process migration easily while still holding the possibility to optimize his code. As stated in section 2.3 there are, however, also problems when using a preprocessor. The problem of debugging user code for example. We believe though, that if a translation scheme between the line numbers in the unmodified code and the modified code can be created, the problem of debugging the user code is not harder than it would be if a preprocessor was not used.

The user does not have to state the variables needed in a checkpoint but merely points in the code where checkpoints should be performed. Data-analysis and insertion of checkpointing code can then be performed by a preprocessor as mentioned in section 2.3.

3.3 Checkpointing

In section 1.3 we stated that the FROST system should support some form of fault-tolerance. The main reason for introducing fault-tolerance in FROST is that the purpose of the FROST system is to perform long term calculations on non-dedicated platforms. The user of such a platform may choose to shut the computer down at an arbitrary time, rendering performed calculations useless unless fault-tolerance is available. Such a feature could be represented by checkpointing. This would not completely remove the problem of performing some calculations twice, but it would help to minimize the problem, as a lesser amount of calculations will be lost.

As process migration basically means moving the state of a process to another computer and resuming its execution, checkpointing could also be used in the process of performing process migration. A checkpoint has to contain enough information from a process to enable the system to recover the process and continue execution after failure. The process migration feature can be implemented by recovering from the checkpoint on a different machine, and continuing the execution there. By using the checkpointing method for saving the state of the process, we can gain fault-tolerance in addition to the process migration feature.

An alternative to using checkpoints for process migration, is to add migration points in the source code as in MpPVM [CS96]. Migration points is different from checkpoints as the process state is only extracted if and when the process is migrated. They give the advantage that when process migration is performed, no calculations are lost. Only if a completely new checkpoint is transferred, this can be achieved with the checkpointing method. As FROST is designed to be used for long term calculations, the time lost by using checkpoints for migration has less influence on the total running time, and is therefore seen as an less important factor. Another advantage with migration points is, that data does not need to be saved to disk before migration is performed. As fault-tolerance is necessary in FROST data must be saved to disk, and this is therefore not considered as a performance degradation compared to the migration point method. Hence, the issues against using the checkpointing feature for migration in FROST is either insignificant with regard to the time lost or unavoidable with regard to saving the data to disk, and we therefore choose the checkpoint solution.

As performing long term calculations is the main purpose of the FROST system it is an advantage to have fault-tolerance as stated above. As stated in section 2.4 fault-tolerance can be secured by saving checkpoints to a stable storage media from where they can be recovered when needed. As the FROST system is meant to run on non-dedicated workstations we cannot be sure that a process can recover even though a checkpoint has been produced and saved to disk. In other words, we cannot be sure

that the disk in a non-dedicated workstation is actually stable storage. This is because a user may choose to remove the FROST system at any time or he may choose to shut down the computer for a long period of time in which the assignment should have been finished. In order to circumvent this problem of users interfering with the normal execution of FROST, some of the checkpoints produced on a slave should be sent to the master of the task for storage until the task has finished. These checkpoints can then be used if a user decides to shut down his computer without announcing it or if a computer breaks down because of an error. Another computer can then be chosen by the master to continue working on the assignment from where the checkpoint was made and less data is lost.

Producing checkpoints in a distributed system is normally an extremely difficult procedure, because, in addition to the normal things included in a checkpoint, communication with other computers across the network has to be taken into account. This means that the system has to support some sort of roll-back feature, allowing a slave to "erase" its previous communication with other slaves. Use of such a feature is potentially problematic as there is a risk that all of the work performed by the system has to be rolled back in order to reach a common ground by each slave.

In FROST we do not have this problem. This is due to the fact that communication in FROST is limited by the way FROST handles the assignments given to it. The flow is that the master splits the assignment into a number of independent tasks. These tasks are sent to slaves which perform calculations and, when these are finished, returns the results to the master. No communication is allowed between the individual slaves. Thus in FROST we do not need a rollback feature because when a task or result has been transmitted it has no effect on the other slaves.

3.4 Process Migration Policies

In the previous sections, the technical issues regarding the process migration and fault-tolerance features were analyzed. As stated in section 2.5 we also need to define certain rules that are used to make decisions when a process is to be migrated. Some information is needed to make the decisions, and a rule must state which information that should be used and how it will be obtained. When that information is obtained, another rule must be used to decide whether we need to migrate a process, and if so, which process we wish to migrate. When a process is chosen for migration, it has to be determined where it should be migrated to. These three rules are known as the information, transfer, and location policies. We have chosen to split the transfer policy into two, creating a selection policy as well. The transfer policy then determines whether it is necessary to migrate a process or not, and the selection policy determines which process to migrate.

In the following sections we will consider each of these policies in greater depth in relation to the FROST system. In order to do this, we will consider three example systems and the policies they use. We will consider the MOSIX system and the Load Sharing Facility (LSF) system described by Milojicic et al. [MDP+00] and tmPVM described by Tan and Yuen [CTY99]. First we will give a short introduction to the three systems, and throughout the analysis of the policies, we will relate FROST to these systems.

MOSIX is implemented as a distributed operating system where processes are migrated between machines in order to perform dynamic load balancing in the system.

LSF provides some distributed operating system facilities on top of various operating systems without changing them. It uses process migration to balance the load in the system, but it is used as a supplement to the initial placement.

tmPVM is an implementation of PVM that supports process migration in a homogeneous environment. Process migration is used to achieve better performance by migrating processes away from overloaded nodes to lesser loaded nodes.

3.4.1 The Information Policy

The information policy consists basically of an information gathering process where information is collected to allow future policies to perform decisions based on the overall state of the entire system. There are many ways of performing this information gathering. They can be centralized approaches

or decentralized approaches, periodic gathering or event based gathering, all depending on the system implementing the gathering of load information. We will not consider how the different systems define load, but how the information is collected.

The MOSIX system uses a decentralized load balancing algorithm where each node in the system maintains a load information vector about the load on a small subset of other nodes in the system. For every iteration of the algorithm two nodes are selected at random and are sent the most recent load information. The receiving system answers this by returning its own load information back to the sender.

The LSF is a little different from the MOSIX system as it primarily relies on initial placement of the processes as a way of achieving load balancing. In addition to this it employs checkpointing as a means of performing process migration. LSF uses a centralized approach where a node is assigned the task of being master. This master is responsible for maintaining the collecting of load information from the other nodes. Each node sends its load information to the master periodically.

tmPVM also uses a centralized approach. A process on each node collects load information, regarding the node it is running on, and sends it periodically to a centralized resource manager.

The FROST system is designed as a peer-to-peer system and a centralized approach is therefore generally an inappropriate solution. With a decentralized approach each machine holds the load information needed to decide whether to migrate or not. If the system are to scale Internet-wide it is not possible for each machine to hold information about all machines in the system. A solution such as the one used in the MOSIX system is required, where only the load of a subset of the nodes is known.

3.4.2 The Transfer Policy

The transfer policy is used to decide whether a system should migrate a process or not. This basically means that this policy only determines whether a node is sufficiently loaded to take action. The policy does not decide which process to move but only whether to migrate or not. We will consider three approaches to this policy, the sender initiated, the receiver initiated and the centralized approach.

The sender initiated policy is invoked on an overloaded node that wishes to transfer a process to another node. According to Milojicic et al. [MDP+00] a sender initiated policy is especially usable in a system where the number of underloaded nodes is higher than the number of overloaded nodes. The receiver initiated approach is in contrast invoked on an underloaded node in need of work. Milojicic et al. says that the receiver initiated policy is good in a system with a larger number of overloaded than underloaded nodes. Shivaratri et al. have a reason for this difference. When using a sender initiated approach in a highly loaded system it is not very likely to find a suitable destination node, and therefore the administration price increases compared to the benefits of migration [SKS92]. Shivaratri et al. furthermore states that when using a receiver initiated approach in a lightly loaded system, the effectiveness is reduced as the need for migration often is discovered late. The centralized approach has a central process that keeps track of the load in the entire system and chooses the nodes that should migrate a process.

The MOSIX system uses a sender initiated approach, deciding to migrate a process when a node finds another node with a significantly reduced load relative to itself. The difference in load between two nodes needed for the transfer policy to make a decision must exceed a stability factor.

If a node is overloaded in the LSF system or if it is needed by a higher priority process, it may choose to migrate a process to another node. The choice of whether to migrate is made from local load information. LSF allows users to specify a certain time period in which local load conditions must remain unfavorable before performing migration on a chosen process. This is done in order to avoid that temporary load spikes can cause unnecessary process migrations. If the load conditions becomes better the process can remain on the current node and the system avoids the time-loss of migrating a process. Depending on the jobs that run in the system, the transfer policy in the LSF system may be configured to use different load information. The LSF system also uses a sender initiated approach.

As described in section 3.4.1, tmPVM has a centralized load information policy, and it is therefore also obvious to have centralized transfer policy. The resource manager keeps track of the load on the

different nodes, and if a load imbalance is discovered, an overloaded node is selected to migrate a process.

As with the information policy a centralized approach is not suitable for the transfer policy in FROST. This is especially true as the load information is kept locally on each node. With regard to choosing a sender or a receiver initiated approach, we need to consider whether the nodes in the system is generally overloaded or underloaded. As we are using non-dedicated machines this property cannot be predicted, and we can therefore not decide which of the two solutions that are most suitable for the FROST system. Instead we will use a combination of the sender and receiver initiated approach, also known as a symmetric transfer policy [MDP+00], as it combines the two policies in order to take advantage of the positive sides in both of them. The sender initiated policy can then be used when the load of a node is above a certain threshold and the receiver initiated policy when a node is below this threshold.

As we use non-dedicated machines such as in the LSF system, the problem regarding temporary load spikes will also exists in FROST. Means must be considered that handle this situation in order not to make unnecessary migrations.

3.4.3 The Selection Policy

The selection policy determines which process to migrate when the decision to migrate a process has been made by the transfer policy. The selection policy is very system dependent. There are many factors that can be used to determine which process to migrate. E.g. a selection policy can choose to migrate the newest process, a long lived process, it can choose based on a the amount of communication a process has with another process on a different node or it can take an entirely different approach to selecting a process to migrate.

In the MOSIX system a process is selected for migration based on its history of forking off new subprocesses or a history of communication with another node if it exists. A process is only chosen for migration though, if it has run for a certain minimum amount of time. This prevents short-lived processes from using up valuable processing time in a migration which turns out to be in vain if the process terminates immediately after it is migrated.

Milojicic et al. [MDP+00] does not describe how a process is chosen for migration in the LSF system.

In tmPVM each node maintains a list of processes that can be candidates for migration. What makes a process a candidate for migration is not specified any further in [CTY99].

In the FROST system the choice of a process to migrate is done locally on each node. This is because we believe that the individual nodes have the most knowledge about the processes running on them. The reason for having a good selection policy is to make sure that as little time as possible is lost when migrating a process. E.g. a process must be running long enough on the (hopefully faster) destination node in order to catch up with the time lost during the migration. In order to achieve this, the selection of a process to migrate could be based on the time that has elapsed since the last checkpoint of the process, the time a process has been running, the amount of data to be transferred, or any other type of information available in the FROST system.

3.4.4 The Location Policy

The location policy is used to determine which node a selected process should be migrated to. Depending on the approach used in the information and transfer policies there can be different possibilities for choosing the destination node of the migrating process. If the load information is kept centralized it is most obvious also to choose the destination node centralized, as all load information will be available. If the load information is not kept centralized and a sender initiated approach is used, it makes most sense to let the source node choose the destination by itself, unless a centralized resource has more knowledge of the nodes in the system, e.g. with regard to architecture or amount of memory. If a receiver initiated approach is used, the location is already determined as the initiator of the migration.

The MOSIX system uses a sender initiated approach and determines where to migrate a process based

on a load vector. If a node is significantly less loaded than the one considering the migration, it can be chosen as a target for migration.

In the LSF system processes may have different requirements, such as special architecture properties, to the nodes on which they execute. These requirements have to be taken into account when choosing the node to which a process should migrate. In addition to this dynamic load conditions are also taken into account when choosing a node. The LSF system has the extra feature that in order to avoid overloading a node, once a process is scheduled to run on a particular node, this node is not taken into account for a period of time when considering where to place new processes. The reason why LSF is able to do this is due to the centralized approach it uses for locating processes.

In the tmPVM system the location policy tries to balance the workload at each node. This is done by migrating processes from overloaded nodes to lesser loaded nodes. For each process that is migrated, the least loaded node is chosen as the destination node, thereby evening out the load on the different nodes.

As stated in section 3.4.2 we choose to use a symmetric transfer policy which includes both the sender and receiver initiated approaches. This choice also influences the transfer policy. When the sender initiated approach is used, a lesser loaded node must be selected as destination node, and when the receiver initiated approach is used, a more loaded node must be selected as source node. The purpose of the location policy in the FROST system is the same as in tmPVM. The policy should be designed to equalize the load on the nodes in the system. This has the effect of making the same amount of CPU power available to each process and thereby letting them execute in the system on an equal basis.

As we wish to make the system as architecture independent as possible, we will not include requirements for special architecture properties when locating a node. If the calculation code must run on a specific architecture, the binary file will only exists for that architecture, and this must of course be taken into consideration when choosing the destination node.

3.5 Demand Specification

In this section we will summarize the choices we have made in the previous sections into a list of demands to the design and implementation of process migration in FROST.

Technical demands:

User space: The process migration feature must be implemented in user space as a kernel space solution does not fulfill our demands.

Indirect extraction: The process state must be extracted using an indirect approach in order to support the heterogeneity of the FROST system.

Preprocessor: In order to uphold the transparency from the users point of view, a preprocessor must be implemented that takes care of modifying the user code to support process migration.

Checkpointing: In order to achieve both fault-tolerance and process migration, a checkpoint solution must be used for saving and transferring the process state.

Furthermore issues regarding transparency, heterogeneity and portability generally have to be taken into consideration during the design process.

Policies:

Information: In order to uphold the peer-to-peer idea in the FROST system a decentralized approach must be used. The scalability of the system must be considered when choosing the approach for distributing the load information.

Transfer: A symmetric approach must be used due to changing loads in the system induced by the local users. Means for securing against process migration due to load spikes must be designed and implemented.

Selection: The process to migrate is selected locally from the information obtained using the information policy. The aim of the selection policy is to choose the most advantageous process to migrate with regard to finishing the task faster.

Location: The symmetric transfer policy also affects the location policy to handle two situations. The purpose of the location policy is to select either a source or destination node of the migration so that the load on both nodes is closer to the average.

When designing and implementing the polices, performance and scalability generally have to be taken into consideration.

PART II

Design

This part deals with the design of process migration in the FROST system. Chapter 4 considers performance aspects that should be kept in mind when designing process migration in FROST. Chapter 5 designs the policies that control the process migration. In chapter 6 the migration procedure is considered. Finally in chapter 7 the preprocessor is designed.

CHAPTER 4

Designing Process Migration

In this chapter we look at the performance issue with regard to policies, checkpointing and process migration that has to be considered before the actual design can start. As the goal of implementing process migration into the FROST system is improving the performance of the system we have to consider how the different parts of the FROST system should behave in order to at least match the performance of the system without process migration and preferably exceed it. In the following section we will consider how this can be done. Afterwards we state the limitations to the present version of the FROST system.

4.1 Performance

In section 1.3 we have chosen to consider improvement of issues regarding load balancing and fault-tolerance and to consider the significance of these improvements with regard to performance in the FROST system. In order to incorporate this into the FROST system it is necessary to consider how improvements such as load balancing in shape of process migration and fault-tolerance in shape of checkpointing influence the performance with regard to the assignments introduced to the system by the users. It is necessary to balance the design of the system so that the maximum performance is gained while still maintaining flexibility with regard to user processes by using process migration. In addition to this the main idea of fault-tolerance has to be considered as well without ruining the performance of the system. The reason for introducing process migration into the FROST system is that we want to achieve better performance when machines are loaded.

In the following we will consider the different areas where performance can be jeopardized by introducing process migration into the FROST system.

4.1.1 Policies

When considering the performance overhead introduced into the FROST system by the effect of the policies introduced in section 3.4, it is vital that this overhead does not grow larger than is absolutely necessary. In order to ensure this we have to consider stability of the policies and the requirement of the policies to make a qualified choice when it is necessary to migrate a process and more importantly, when it is not. These aspects will be considered next.

Stability

Stability is an important aspect of the policies in the FROST system. The policies must be stable so that no unnecessary process migrations happen and that the system is not subjected to thrashing. It is very important that both of these two unwanted properties does not happen in the system as they have a direct impact on the performance of the system. An unwanted migration is if a process is migrated to a node that is already heavily loaded. The system would not gain anything from performing this migration and therefore it makes no sense to migrate the process. The same goes for thrashing. Thrashing happens when a process is migrated back and forth between nodes in the system without ever having an opportunity to perform any calculations. In order to avoid this the policies must ensure that when moving a process the state of the system after the migration is always better than it was before the migration. This will result in a better overall load in the system and ensure that performance is kept at a maximum with the properties given by the system.

The state of the system after a migration has been performed is very important. If the migration just moves load from one loaded node onto another which becomes equally loaded after the migration,

nothing is gained from the migration and it is more likely that time is lost in the migration. This situation is an unwanted migration which leads to instability as unnecessary migrations will occur. The problem of maintaining stability is that in order to do so the system have to predict the state after a migration is performed, but before it actually happens. This must be done to prevent the migration from happening in the first place if it turns out that the result of it is a worse system state than if the migration is not performed.

4.1.2 Process Migration Overhead

When the FROST system is making a choice whether to migrate a process or not there are a number of parameters that should be taken into account. Among these are the amount of time that is lost when a process is migrated.

It is vital to the performance of the system that the relation between the overhead for creating a checkpoint and the time between the checkpoints is tuned so that as little calculations as possible is lost at system failure, without inducing too much overhead. In addition to this we have to ensure that the amount of overhead that is induced by the checkpointing and process migration into the overall time is also kept at a reasonable level.

In the following we will consider ways of optimizing the overhead induced by checkpoints and process migration.

Checkpointing

In order to limit the overhead introduced into the system by the time that is lost when a checkpoint is migrated we have to consider the optimal time between checkpoints. If this can be tuned properly the system will loose as little calculations as possible in relation to the time it takes to perform a checkpoint.

A checkpoint is a point in time where the state of a process is saved as described in section 3.3. Therefore we loose the calculations that have been performed since the last checkpoint as they are not part of the checkpoint used for recovery. In order to boost performance we have to minimize the amount of calculations we have to perform twice. Therefore we have to weigh how often we wish to create a checkpoint against the cost of making a checkpoint and the probability that a computer crashes. The problem is that the FROST system cannot predict how big a checkpoint will be and how long it will take to make it, as it depends heavily on the assignment created by the user. It is up to the user to know the overhead of checkpointing and the probability that this computer crashes. It could be argued then that it could be an advantage to create checkpoints with very small intervals but the time between checkpoints must not be too small either as the smaller this interval gets, the more overhead is induced into the overall computation time when performing checkpoints. Plank and Elwasif [PE97] gives an equation for calculating an approximation of the most optimal space between checkpoints. This is shown in equation 4.1.

$$T_{opt} = \sqrt{\frac{2C}{\lambda}}, \tag{4.1}$$

where C is the overhead induced by creating a checkpoint and λ is MTTF^{-1} (Mean Time To Failure). If for example we consider a machine that has an MTTF on 4 hours and the checkpointing overhead, C, is 2 seconds, then the approximated optimal time between checkpoints is 4 minutes. The average tasks is supposed to run for hours in FROST and therefore 4 minutes is a reasonable loss if a computer breaks down.

Migration

As FROST is a distributed system that operates in a non-dedicated environment we have to consider that the load on machines can change over time. This means that if the load increases on a computer it cannot provide the same performance as before the change in load and it might be advantageous to move a process to another computer that provides better performance. In order to loose as little

calculations as possible it might be advantageous to reconsider the time between checkpoints calculated above. This is due to the fact that we expect changes in the load on a node to occur more often than break downs of the nodes. Instead of MTTF as a measure for the probability λ we can now use the probability that the load will change and cause a migration to happen as λ. The probability that a migration happens is based on the behavior of the users that use the computers in the FROST system for regular use. If a user chooses to use his computer heavily it may trigger a migration. In order to decide this probability it is necessary to study the usage patterns of users. As this is beyond the scope of this project we choose instead to checkpoint more often in order to loose as little calculations as possible. We assume that every 30 minutes the user of a machine will use it intensively long enough to justify a migration. This changes the optimal checkpointing interval calculated using equation 4.1 to a checkpoint every 38 seconds.

Necessary Variables

In section 4.1.1 it was stated that the time it takes to perform a checkpoint depends on the amount of data which is saved in a checkpoint. In order to maximize performance with regard to the time it takes to perform a checkpoint it is necessary to limit the data in the checkpoint to as little as possible. As stated in section 3.3, a checkpoint has to obtain enough information about a process to save it and recover it at a later time, either on the present node, or on another node if the checkpoint is used in a process migration. The data that needs to be saved can include a number of things, such as variables, and pointers to other objects. In order to optimize the checkpointing process an approach resembling the one used by Chanchio and Sun [CS96] in MpPVM should be considered. As stated in section 2.3 MpPVM only includes what is called *necessary* variables when a migration of a process is performed. This ensures that as little data as possible is stored and therefore as little time as possible is used in the migration phase.

4.2 Limitations

In the following we will state the limitations we set from the analysis to the design.

4.2.1 Scalability

The FROST system is supposed to supply a large amount of processing power at a very low cost by using a large number of computers. The present version of the FROST system is only running on a small number of computers in a LAN and we will in this project continue to consider the including of process migration in FROST when used on a local area network. This also means that the solutions provided in the design does not necessarily scale well to an Internet wide platform. We are aware of the limitations this introduce into the system but we will defer the discussion of scalability until a later chapter.

4.2.2 Necessary Variables

As described above, the locating of necessary variables will make it possible to achieve better performance as a minimum of variables are checkpointed and transferred during migration. Locating necessary variables does, however, demand a high level and very complex code analysis. Furthermore it is not a necessary feature to support in order to achieve a reasonable process migration feature, and we therefore choose not to use this approach to variable extraction in our checkpointing procedure.

4.2.3 Fault-tolerance

Fault-tolerance is an important feature in a vulnerable environment. In this project, however, we have chosen to concentrate on the process migration feature and we will therefore not design the fault-tolerance features any further, except for the checkpointing feature which is used to extract the state of the processes.

CHAPTER 5

Policies

In section 3.4 of the analysis we considered four policies for controlling the process migration procedure. These were the information policy, the transfer policy, the selection policy, and the location policy. These policies are essential parts of a system which incorporates process migration and therefore we will in the following sections consider how they can be included into the FROST system.

5.1 Information Policy

The information policy states how the system is to gather information on which to base later decisions. In order to do this we have to consider what this information is to consist of, how it should be gathered and how it is published so that all nodes in a network can gain access to this information. In the following sections we will consider these issues in greater detail.

5.1.1 Usable Information

When considering a system as FROST, we see that its main limitation with regard to obtaining CPU-cycles is that it must yield whenever a user wishes to use the computer. This means that the FROST system cannot monopolize a node and shut out all other programs on that node in order to finish a result as fast as possible. As the nodes are non-dedicated the operating system on each node must ensure that a user has priority over the FROST system. If a user is pushed aside by FROST the entire idea of the FROST system is jeopardized.

A successful FROST node is a node that performs its tasks as fast as possible and still offers a good response time to the local user of the node. The problem is how to secure the responsiveness of a node. As stated in section 1.3 this is secured by giving a FROST task a low priority in the operating system. This priority is lower than the priorities assigned to the user processes in that system thus ensuring that the FROST system is preempted whenever a user process needs the system.

A usable variable in the information policy is the load on a node. This can be measured in several ways, an example is the number of processes in the ready queue - the more processes in the ready queue the more loaded the system. Kunz states that using the ready queue as a measure for the load of a computer is the most effective way of measuring load [Kun91].

We choose to use another way of measuring the load on a node. As the processes in the FROST system are assigned the lowest possible priority in order not to interfere with user processes, we believe that the ready queue does not supply sufficient information for calculating load. The problem is that as long as there is user processes, the FROST process will receive little CPU time as it has the lowest priority even though the ready queue is short. This means that the length of the ready queue does not supply usable information for the FROST system as the length may only be one, namely the FROST process, and instead of taking the consequences and migrating the FROST process, we do nothing based on the short ready queue leading to starvation of the FROST system. In order to avoid this starvation of the FROST processes we choose to use the available resources as a measure of performance. The available resources states how much processor time is available to the FROST process and thus whether it is advantageous to move the process somewhere else in order to allow the FROST process to run. By using available resources we are always able to find the node that finishes the assignment in the shortest time. Resources is an independent measure where load is relative to the node on which it is taken. If for example both a slow and a fast node is 80% loaded, the fast node will have more available resources as each percent count for more on the fast node. In other words percentages cannot be compared directly. The number of available resources on the other hand can always be compared between machines no matter how fast they are relative to each other. The

approach that we have chosen to include in the FROST system is to execute the following method at regular intervals.

CALCULATERESOURCES()

```
1: set timer
2: while the timer runs do
3:     CurrentResources ← CurrentResources + 1
4: end while
5: return CurrentResources
```

The intervals that the above method should be run with depends on two things. First of all the more often it is run, the more often resource information is obtained. The problem with obtaining data often is that it comes with a price. The more often the CALCULATERESOURCES method is executed, the more processing power is used in calculating resources and the less processing power is available for the calculation of tasks in FROST. The second thing that the interval depends on is how often migration can be considered. If the price of migration is too high the interval with which it is called should not be very high.

The timer that is mentioned in the description of the CALCULATERESOURCES method decides for how long the single measuring of available resources will last. The value measured will be passed on to the following policies for further decision making. The timer plays an important role as well as the longer it runs the less sensitive the policies will be toward fluctuations in the resources. But if it runs for too long, the same problem as with the intervals happen. Too much time is spent in the CALCULATERESOURCES method.

The CALCULATERESOURCES method is a rather simple method but it has no need to be more complex in order to give a measure for the available resources in the system. The timer which is mentioned in the above method stops the while-loop, ends the CALCULATERESOURCES method and a measure for the available resources in the system is returned to the caller.

The CALCULATERESOURCES method must be run at the same priority level as the processes which performs the calculations. This will then give an average measure for the amount of resources available for each process and this number can be compared between machines.

As FROST uses non-dedicated machines, the amount of memory is very likely limited compared to when a supercomputer is used. For this reason the amount of memory should also be considered in the available resources. The amount of free memory and the CPU usage can change rapidly, but if the load becomes too high, calculating processes can yield, giving more CPU power to the user. Memory on the other hand has to be swapped out thereby making more available memory to other processes. This however, takes some time as it requires disk access and it cannot be controlled by FROST, as it is only operating in user space[1].

We have chosen a simple solution to the problem of handling memory resources. We have chosen to limit the use of memory to a predefined value, either set by the user or as a percentage of the total amount of memory. This value determines the pool of memory that may be used by calculating processes. As designed in [GK02], the user must specify the maximum amount of memory used by each work unit. We believe this is a reasonable demand, as the amount of memory used by an algorithm most often can be determined on beforehand. When a work unit is sent to a machine, the amount of memory it uses is subtracted from the pool of available memory. If there is not enough available memory in the pool, the work unit cannot be sent to that machine. The amount of available memory must be declared when a machine requests to transfer a process to another. This can be handled by sending the memory use of a process along with the request. The receiver can then make the decision whether it has sufficient memory to accept the process.

5.1.2 Gathering Local Information

In the previous section we stated what kind of information we could include in the information policy and how this information could be obtained. We also have to consider how to enable the system to obtain it so that we display a fair picture of the load in the system. Locally on each machine we have

[1] FROST cannot control the yielding of processes either, but the calculating processes can be set to run with a low priority.

to perform resource calculations periodically in order to discover if the available resources changes for example because of the user. Event based calculations are not adequate as we cannot count on any events happening frequently enough in the FROST system. It is obvious that if the resource calculation runs too often, too much time will be spent updating the available resources information across the network and less time is used for calculating the user tasks. But if the resource calculation are performed too infrequently a node may choose to move a process based on old information which is no longer valid, resulting in overloading an overloaded node even further.

Another problem, first introduced in section 3.4.2, that we have to consider is the problem of avoiding reactions based on spikes in the values of the resource calculation. In the LSF system [MDP+00] this problem was addressed in the transfer policy. In FROST, however, we see that it is advantageous to solve it in the information policy as it is here the gathering and processing of information is performed. In order to smoothen out load spikes in the system, we choose a solution where resource information is gathered over a period of time and an average value of the collected information is calculated. If the period is chosen correctly the influence of load spikes can be minimized. In addition to this we choose to require that two gatherings of resource information has to show the same tendency in available resources in the system before it is considered to take further action in order to minimize the sensitivity of the system against short lived processes.

In order to consider the interval with which the resources should be gathered on a node we have to take the non-FROST processes of the users into consideration. As stated above we require that two intervals of resource checks show the same tendency before a process is migrated. We do not wish the system to be sensitive to processes that run for less than one minute which means that we have to perform the resource gathering with a one minute interval.

It should be noted that the approach of gathering information should be tested and adapted to performing the resource calculation with the most optimal interval.

5.1.3 Distributing Information

The problem of introducing process migration into a system is that one must be very careful not to move a process from one node which is overloaded onto another node which is even more loaded. In Shivaratri et al. [SKS92] they present an approach where nodes are chosen and polled for their load based on their previously known state. In FROST, we think, it is a better approach to share load information among the individual nodes in the system. Sharing load information is relatively easy as long as the system is sufficiently small. It is easily done by sending the load information of one node to all the other nodes in the system, either using unicast, broadcast or multicast. The advantage of this approach is that a node can easily be chosen as target for migration or request as all load information is known on beforehand. This also enables us to make a preliminary prediction about whether moving a process would be advantageous or not. Prediction will be considered later. The problem with this approach is that it cannot scale to a size where it is usable on a system spanning the entire Internet due to the large amount of information that each node should contain. Another problem is that broadcast is usually not allowed to cross routers between a sub-net and the Internet, multicast requires that all nodes agree on the address, and unicast requires that all nodes has knowledge of all other nodes.

The FROST system is running on non-dedicated computers using a non-dedicated network for communication and therefore we cannot rely on a single server to perform information gathering and distribution so we have to consider another way of upholding the information policy. As stated in section 4.2 FROST is presently only running on a small number of computers. Therefore we have chosen to broadcast the resource information of each computer so that all other computers gain access to this information easily. We are aware of the problems connected with this approach and it will be necessary to consider an approach which avoid the use of broadcast in order to allow the FROST system to scale. This will be considered later.

Each node in the FROST system is a potential master and as stated in section 3.4.1 all nodes function in a peer-to-peer fashion, sharing processing power on an equal basis. As every node functions on an equal basis it is necessary for each of them to have a local data structure for containing the resources of all the nodes in the system, including the local node. We will call this data structure a $ResourceVector$ and it must be present on all the nodes in the system. The reason for saving ones own

resource information as well as the resource information of all other nodes, is that the information is used in the transfer policy when thresholds are calculated. As described previously in sections 5.1.1 and 5.1.2 the resources on each machine is calculated regularly by the master and shared with the other nodes in the system.

Because resource information of a node has to be shared with all other nodes in the FROST system, we have to ensure that only essential information is shared so that nodes are not overwhelmed by a huge amount of resource information. It is for example not necessary to share newly calculated resource informations if these are equivalent to the old information which has already been shared. Therefore we choose to use an event based distribution scheme, based on the difference between old and newly calculated resources for the information which has to be shared with other nodes. Such a scheme prevents sharing of resource information in the FROST system unless it is sufficiently different from previously shared information.

When sharing resource information we will use the approach shown in algorithm 1.

Algorithm 1 Information policy.

1: $CurrentResources \leftarrow$ CALCULATERESOURCES
2: **if** $PreviousResources$ is very different from $CurrentResources$ **then**
3: $ResourceVector[Localhost] \leftarrow CurrentResources$
4: $PreviousResources \leftarrow CurrentResources$
5: Broadcast $CurrentResources$
6: **else**
7: $ResourceVector \leftarrow CurrentResources$
8: **end if**

Whenever a master receives new resource information from a node it updates its $ResourceVector$ to reflect this information.

This approach is usable when the system is running and every master has information about all the other masters in the system. But if a new node is introduced in the network special measures has to be taken to ensure that a new node can participate on the same terms as the others. This approach is shown in figure 5.1. When a new node is started in the FROST system, it immediately announces its presence to the other nodes by issuing a broadcast message containing information about itself, including its current resource information. After receiving this message each of the other nodes waits for a random amount of time and unicasts a message to the new node, containing similar information about itself. The new node is then updated and can now begin calculating tasks. The reason that the nodes have to wait a random amount of time before answering the new node is that if they did not wait it could result in the new node being overwhelmed by the resource messages as they would arrive more or less simultaneously.

5.2 Transfer Policy

The transfer policy constitutes the decision making part in the system. It is the transfer policy which has to make the decision about whether the system is in a state where it is necessary to take action in order to get out of that state.

As stated in section 3.4.2 we use a symmetric approach for deciding whether to receive or send a process. A symmetric approach is a combination of two parts, a sender and a receiver initiated approach. The sender initiated approach is used if a computer has few available resources. Then it may decide to transfer one of its processes to another computer in order to balance the overall resources in the system. As the main purpose of the transfer policy is to get the most calculations out of the resources present in the system, we must ensure that no node is able to exploit other nodes. This is done with the peer-to-peer structure of FROST in mind and is ensured by the location policy. The receiver initiated approach works opposite of the sender initiated approach. If a computer has a large number of available resources it can request a process from another computer in order to lessen the load on that computer.

The decision about whether to use the sender or the receiver initiated part of the symmetric approach

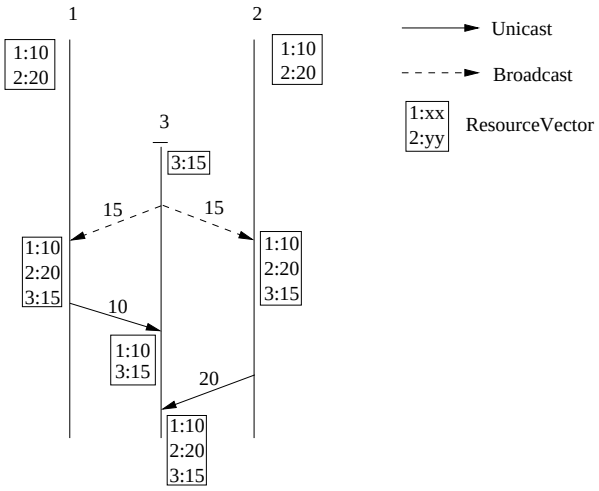


Figure 5.1: The procedure of sharing resource information with a new node. Nodes 1 and 2 are already online nodes, node 3 is the new node. When node 3 is started it only knows about itself. It then issues a broadcast with its own resource information. Nodes 1 and 2 receives the broadcast and updates their information. They then wait a random time before they unicast their own resource information to node 3.

is made by the individual nodes in the system, based on the average available resources in the entire system. In the following section we will consider how to make this decision.

5.2.1 System Thresholds

When it comes to comparing the available resources on a local node with the available resources in the entire system the solution that we choose in the FROST system is to use three states delimited by two thresholds, an upper and a lower threshold. Shivaratri et al. describe a similar approach but where they calculate thresholds based on the local load only [SKS92], we decide to calculate them relative to the average available resources in the entire system because we believe that this gives a more useful image of the resources in the system. These thresholds state whether the system is loaded or not and they determine the eagerness of the system with regard to migrating processes. If the resource-number is above the upper threshold the system is lightly or not loaded and the transfer policy may choose to request a process from another node which has a lower resource-number than itself. If the resource-number is below the lower threshold the system is heavily loaded and the transfer policy will try to migrate a process away in order to make the system less loaded. If the resource-number is between the two thresholds no action will be taken. The closer the thresholds are placed together the more eager the system is to move processes.

The reason for using the thresholds is, in addition to using them for deciding whether the system is over- or underloaded, that the system is not as sensitive with regard to fluctuations as there is some freedom of movement for the resource-number to move around the average value before a process is migrated if they are placed with a reasonable interval. If for example thresholds were not used, a node would choose to migrate a process as soon as its available resources fell below the average system value and that would likely result in thrashing.

We choose to fix the thresholds to a certain value around the average available resources in the entire system. This is done in order to give the nodes a certain leeway before they choose to migrate a process. In addition to this we choose to switch off either the receiver or the sender initiated approaches at certain points, due to the fact that their advantages lie in different ends of the resource spectrum, as stated in section 3.4.2. An example of this can be seen in figure 5.2

In figure 5.2 (a) we see an example on a system which has a high number of average available

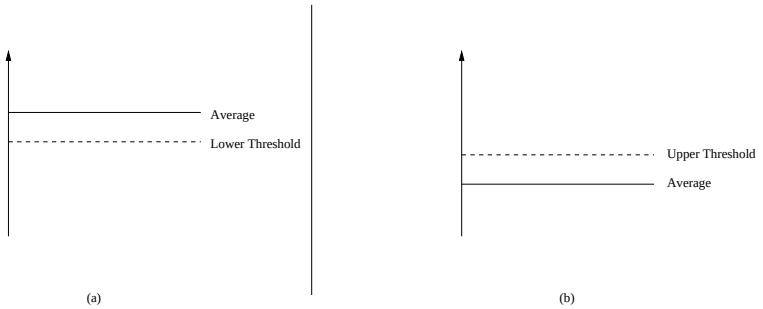


Figure 5.2: An illustration of different approaches.

resources. As can be seen, only the sender initiated approach is used, represented by the lower threshold, as it is most effective when the number of underloaded nodes is higher than the number of overloaded nodes in the system. In figure 5.2 (b) it is the other way around. Here a small number of underloaded nodes are available in the system and thus a receiver initiated approach is more effective, represented by the upper threshold. The transition between a sender and a receiver initiated approach should happen simultaneously on all nodes in the system in order to ensure that no conflicting requests are issued. This would require that all nodes in the system had a shared state which could be set according to the current average available resources in the system. Creating such a shared state is complicated and would result in a significant communication overhead in order to synchronize all nodes.

Another possibility is that each node measures two extremities, a number for the available resources when the node is idle and a number for the available resources when a node is heavily loaded. These values shall only be measured once, namely during the installation of the FROST program. Every time a node is going online the values are broadcast as part of the first broadcast message which is described in section 5.1.3. The extremity values can then be used to calculate an average idle value and an average heavy-loaded value for the entire system. Using this information each node can make a choice whether to use a receiver or a sender initiated approach - if the system average available resources is closest to the average idle value a sender initiated approach is used and the receiver initiated is switched off, and vice versa. This approach is illustrated in figure 5.3.

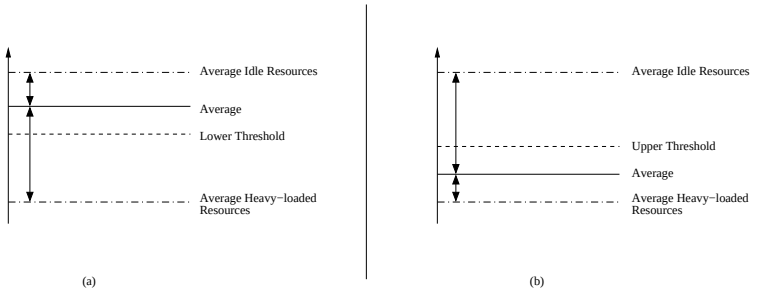


Figure 5.3: An illustration of how the system makes the choice between a receiver and a sender initiated approach.

As can be seen in figure 5.3 (a) the distance from the average available resources to the average idle value is shorter than the distance from the average available resources to the average heavy-loaded resources. This results in the system choosing a sender initiated approach as there are fewer nodes which has few available resources and thus a sender initiated approach is more effective. The opposite can be seen in figure 5.3 (b) where the average available resources in the system is seen to be closer to the heavy-loaded resources extremity, due to a lower number of nodes with a lot of available resources. This results in a receiver initiated approach as it is better that the few nodes with a lot of

available resources ask a node with a few number of available resources for a process than if all nodes with a few number of available resources just send their processes to the nodes with the high number of available resources. We still use the thresholds to determine when a migration shall be initiated.

If a new computer joins the system it receives values for available resources, and idle and heavy-load extremities. This means that it can immediately calculate the average available resources in relation to the extremities, and determine whether a sender or a receiver initiated approach is currently used in the system. If a sender initiated approach is used, it will do nothing until it is contacted by a node which has few available resources, and thus is placed below the lower threshold. The node which has few available resources states a wish to migrate a process to the new node, which the new node can accept or deny depending on various parameters which will be discussed later. If a receiver initiated approach is used, the new node will find a node which has the least available resources and request a process from that node. Again there are certain parameters that must be fulfilled before migration is initiated.

Determination of the thresholds is a subject that should be looked into in order to optimize process migration in the FROST system.

Determining System State

When determining the system state we have to consider the fact that we cannot calculate the average available resources in the system using resource values obtained at the same time for all the nodes in the system. This is primarily due to the lack of a shared clock between the nodes for timing the checking of resources so that every node checks its available resources at the same time. This means that the average available resources calculated in order to determine the system state is not the average available resources in the system at a fixed time, but an approximation using the available values for the nodes in the system. These values for the available resources may be far from the actual values at the time of threshold calculation, but it is the best approximation which can be reached in a realistic manner. It would be possible to obtain a more exact number by either using a snapshot algorithm or synchronizing the clocks in the system, but this would increase the cost of determining the system state in a way which is undesirable.

In algorithm 2 we give the algorithm for calculating the system state in the FROST system and how we act upon this information. Note that action is only taken if two resource values obtained after each other shows the same tendency. So if the $PreviousResources$ states that the system state is above the upper threshold and the $CurrentResources$ states the opposite, then no action is taken.

5.3 Selection Policy

The selection policy is the part of the system that makes the choice of which process on a node that should be selected for migration. In section 3.4.3 we have argued that the selection policy is performed locally because the information which is required for making the decision is only known locally. We also stated that the process which should be chosen for migration should be the one which was most advantageous with regard to finishing faster. In order to make the choice we gave some parameters. These parameters were the following.

- The time a process has been running.
- The amount of data that must be transferred during migration.
- The time since the last checkpoint was performed.

In order to make the best choice of which process to move all the above parameters can be used in the selection process by weighting them against each other. This can be done by introducing a score system where each process receives a number of points for each of the parameters. The score will depend on how a process is situated in relation to the other processes and in relation to the demands set by the parameters. This will then allow a node to choose the most optimal process for migration.

Algorithm 2 Transfer policy.

```
1: if broadcast with resourcenumber is received then
2:     ResourceVector[IP] ← resourcenumber
3:     Calculate new average available resources in the system
4: end if
5: if average available resources has changed enough then
6:     Calculate new distance from average available resources to extremities.
7:     if distance to idle extremity is smallest then
8:         SystemMode ← SENDER
9:     else
10:        SystemMode ← RECEIVER
11:    end if
12: end if
13: ...
14: if SystemMode = RECEIVER then
15:    if ResourceVector[Localhost] > UpperThreshold then
16:        if PreviousResources > UpperThreshold then
17:            The node is lightly loaded and should request a process
18:            goto locationpolicy
19:        end if
20:    end if
21: else if SystemMode = SENDER then
22:    if ResourceVector[Localhost] < LowerThreshold then
23:        if PreviousResource < LowerThreshold then
24:            The node is loaded and should get rid of a process
25:            goto selectionpolicy
26:        end if
27:    end if
28: else
29:    Do nothing
30: end if
31: PreviousResource ← ResourceVector[Localhost]
```

The procedure of selecting a process for migration consists of two parts - calculating scores for all the processes on the node based on the criteria mentioned above, and making a choice, based on this score, about which process on a node to migrate. This is approach is shown in algorithm 3.

Algorithm 3 Selection policy.

1: **for all** processes on the node **do**
2: Calculate score based on the criteria given
3: **end for**
4: Choose the process with the best score for migration

There is a problem with using the score system for deciding which process to migrate though. In order to ensure that each of the parameters are upheld the score has to be kept up-to-date. As some of the parameters may change in a way which cannot be foreseen, the score assignment is a dynamic feature of the system as scores must be reassigned every time a process is to be chosen for migration.

5.3.1 Score Calculation & Process Selection

The primary goal of the score calculation is to allow the selection policy to choose the process which is the most advantageous for migration. As we stated above the score must be recalculated whenever a process is to be selected in the FROST system due to the nature of the parameters on which the score is based. In order to select the most optimal process a lot of things can be taken into account.

One approach is to weight the individual parameters in the score up against the others so that some parameters have more significance than other parameters. For example it is obvious that the time since the last checkpoint plays a major role in score calculation as the longer it has been since the last checkpoint the more data is lost if it is used to restart the process. Therefore it could be an advantage to let this parameter have greater influence on the final result when the score is calculated. Another important aspect that could be included in a weighting procedure is the size of a checkpoint. It is necessary to take the time it takes to move a checkpoint from one computer to another into account as well when considering how to weight parameters. If for example a checkpoint is 30MB in size it could be a problem to move such a checkpoint no matter how often the checkpoint is performed. The time it takes to move 30MB depends heavily on the speed of the connection from the sending node to the receiving node. If it only takes 30 seconds to move the 30MB and the process has a 4 minute old checkpoint, there will be lost less than 5 minutes if the process is migrated. But if it takes 10 minutes to move the 30MB, approximately 15 minutes of calculation time is lost, and in addition to that we have loaded the network for 10 minutes. Therefore it will be a good idea to weight the scores so that we find the process which looses the least time when it is migrated. This example also indicates that the network bandwidth should be taken into consideration both when the destination machine is selected and when the process to migrate is selected. As shown above, the network bandwidth can increase the migration time considerably.

In section 3.4.3 we chose to include run-time for a process, time since latest checkpoint, and amount of data that is to be transferred in our selection procedure. A possible procedure for calculating the score in the FROST system, based on the three the parameters given above, is as follows.

Run-time for processes: In the FROST system we assume that processes in the same overall assignment generally will run for the same amount of time. In order to have a measure for the expected remaining running time, an average can be maintained for each of the assignments that a node is involved in. Then a score can be assigned by comparing the elapsed run-time and the expected run-time for a work unit. The maximum score is given to the process which seems to have the best potential of continuing for the longest amount of time. This ensures that a process which is just about to finish will have a lower possibility of being migrated. The second highest score is awarded to the process which will continue second longest and so on down to the process with the shortest expected running time left. The reason why it is the processes which will continue for the longest time that gets the highest score is that it has the best probability for catching up with the time lost by migrating.

Amount of data transferred: As noted above, the amount of data that needs to be transferred can

have a very large impact on the time lost by the migration. The least amount of points should be assigned to the process that has the largest amount of data to transfer. The amount of data should be considered in the assigning of points, e.g. by assigning one point per 100kb of data. This will give the process with the most amount of data most points, which is not wanted. To invert the points, each assignment can be subtracted from the maximum amount of points given, thereby assigning zero points to the process with most data. The points should be weighted with regard to the bandwidth.

Time since last checkpoint: Each time a process makes a checkpoint it also records the time. When scores are assigned, it is possible to calculate the time that has elapsed since the last checkpoint for all the processes. The above procedure for inverting the point assignment can also be used here. The process with the youngest checkpoint should get the highest score as the least amount of time will be lost when recovering from it. If one point is assigned for each minute since the last checkpoint, the oldest checkpoint is assigned the highest score. By inverting the scores, the desired assignment is achieved.

The scores are summed up to a total and the process with the highest score is to be migrated. An example is given in table 5.1. The amount of data to transfer includes the size of the binary file.

	Process A	Process B	Process C	Process D
Expected run-time	30 min = 2pts	2 min = 1pts	77 min = 3 pts	120 min = 4pts
Data to transfer	2 Mb = 0pts	1,2 Mb = 8pts	1,7 Mb = 3pts	1,8 Mb = 2pts
Time since last checkpoint	2 min = 13pts	15 min = 0pts	10 min = 5pts	5 min = 10pts
Total	15pts	9pts	11pts	16pts

Table 5.1: An example of the score assignment process.

By comparing the result from table 5.1 with the price of the migration shown in table 5.2, we see that some weighting of the points assigned would be an advantage. With a 10Mbit connection it is clear that process A would be more advantegeous to choose than process D, as only 2 minutes is lost. If, however, a 56kbit connection is used, it may be an advantage to transfer process D, as it has an expected running time of 120 minutes left to catch up with the 9 minutes lost. Process A has only 30 minutes to catch up with the 7 minutes lost by choosing that process.

	Process A	Process B	Process C	Process D
10 Mbit	2 min	15 min	10 min	5 min
128 kbit	4 min	16 min	12 min	7 min
56 kbit	7 min	18 min	14 min	9 min

Table 5.2: The amount of time lost by migrating a process. Only the time since last checkpoint and the transferral of data is considered. The time for transferring data is calculated using theoretical maximum values for the connections.

From the examples we can conclude that the higher bandwidth available, the more the time since last checkpoint should be weighted compared to the amount of data to transfer. Furthermore it would be an advantage to compare the time lost with the expected run-time e.g. taking the expected available resources on the destination machine into account. As can be seen from the examples, the use of a scoring system is rather complex, as the weighting of scores should be performed dynamically taking the bandwidth and available resource etc. into account.

According to Eager et al. [ELZ86] it is often as good a choice to use a simple approach when performing load balancing as it is to use a highly complex approach. Process migration is essentially dynamic load balancing and therefore we have chosen to limit the scoring system to use a simple approach. At present FROST is run on a LAN and therefore has a high amount of bandwidth at its

disposal and therefore we find it reasonable to omit this parameter. We choose to use a single value for the selection of a process to migrate: The time since last checkpoint. Hence, the process with least time since the last checkpoint was performed is chosen for migration.

5.4 Location Policy

The location policy is the mechanism that chooses a destination for a process that is to be migrated. As stated in section 3.4.4 we have chosen a symmetric policy. This means that depending on whether the receiver or the sender part of it is used, the location policy has to find the least or the most loaded node in the system.

If the system is in the receiver mode the location policy has to find the most loaded node in the system. This is done by looking through information about all nodes in the system and choosing the one which has the fewest available resources. That node is then offered the possibility of offloading a process onto the initiating node. In the receiver initiated approach the upper threshold mentioned in section 5.2 will only be used by the requesting node to check whether it is in a position to allow a request, as the requested node does not need to be overloaded but only more loaded than the requesting node. The algorithm used in the receiver initiated approach is shown in algorithm 4. The node variables are data structures containing information about a node, such as the IP and the available resources.

Algorithm 4 Receiver initiated location policy.

```
1: ChosenNode ← 0
2: for all Nodes N in ResourceVector do
3:     if N.resources < ChosenNode.resources then
4:         ChosenNode ← N
5:     end if
6: end for
7: Offer to receive a process from N.
```

In a sender initiated approach a node chooses to offload a process if possible due to lack of available resources. This choice has been made in the transfer policy, and a process has subsequently been chosen for migration in the selection policy. It is now up to the mechanisms in the location policy to choose a new node to which the process is to be migrated. This is done by looking through all the nodes and choosing the node with the most available resources as a target for the migrating process. This choice can only be made by nodes which are below the lower threshold, which denotes a heavily loaded system, in order to secure that the load is balanced across the entire system. If a non-overloaded node is not found the chosen process is restarted and the migration is not performed. The algorithm used in the sender initiated approach is shown in algorithm 5

Algorithm 5 Sender initiated location policy.

```
1: ChosenNode ← 0
2: for all Nodes N in ResourceVector do
3:     if (N.resources > lower_threshold) then
4:         if (N.resources > ChosenNode.resources) then
5:             ChosenNode ← N
6:         end if
7:     end if
8: end for
9: if ∃ ChosenNode then
10:    Send a process to it.
11: else
12:    Skip migration for now.
13: end if
```

There is a problem with these approaches as we run the risk of migrating a process to a system which will become heavily loaded when receiving a process. This force the receiving node to migrate a

process, and so on leading to thrashing. A solution to this problem is to only migrate processes to nodes which are above the upper threshold. Shivaratri et al. [SKS92] propose a prediction method where a migrating node polls another node at random for its load. Then the migrating node determines whether sending another process to the chosen node will result in making the load exceed a certain threshold. If it does, another node is chosen at random and tested. If not, the migrating node sends a task to the chosen node. In the example described by Shivaratri et al. they do not have local access to the load information of other nodes and thus they have to poll for it. In FROST, on the other hand, we have information about the current available resources of all nodes in the system. This can be used to introduce a prediction approach in FROST when a process migration is about to occur. If the prediction is positive the process is migrated, if it is not, no migration is performed to that node. This prediction approach will be considered next.

5.4.1 Prediction

In order to balance the resources in the system in the most optimal way there must be some overall rules for deciding when it is a good idea to migrate a process and when it is not. These rules are to ensure that the resource situation is always better in the system after a migration has been performed than it was before the migration. This can be done by demanding that the receiver of the migration has a sufficient amount of available resources and that the available resources of the two machines must be closer to the average available resources in the entire system after a migration from one of the machines to the other has been performed.

Calculating Needed Resources

The primary goal of migrating a process is to choose a node which offers to finish a task faster than the one it is presently running on, or in other words, the node which has a sufficient amount of available resources. Using the time that is lost when migrating a process we can consider the minimum amount of available resources that should be available for a process when it has migrated to a new node for the migration to be advantageous. This estimation is only performed in the sender initiated approach. It can be used by the sender initiated approach to choose the node which is most likely to finish a task chosen by the selection policy fastest. Calculating this can be done by looking at the relation between the remaining time for a work unit and the available resources on the current node. In addition we have to take the time it takes to migrate a process into account. This is seen in equation 5.1.

$$\frac{P_{rem}}{R_1} > \frac{P_{rem} + T_m}{R_2}, \tag{5.1}$$

where P_{rem} is the remaining time for a work unit, R_1 is the current available resources on the old node, R_2 is the resources that must be available on the new node, and T_m is the time that is lost when migrating a process.

The time that is lost when migrating a process can be calculated as in equation 5.2.

$$T_m = \frac{S_c}{n} + T_r + \frac{1}{2} \cdot P_c, \tag{5.2}$$

where T_m is the time that is lost when migrating a process, S_c is the size of the checkpoint, n is the speed of the network, T_r is the time it takes to recover a process from a checkpoint, and $\frac{1}{2} \cdot P_c$ is the average computation time that is lost when moving a checkpoint.

Equation 5.1 state the amount of resources that a node must have in order to receive a process. This calculation can be used to limit the number of nodes that are considered when the next phase in the prediction approach is initiated, namely finding a node that can ensure that the available resources of the two machines are closer to the average available resources after the migration has been performed.

Predicting Resources

In figure 5.4 an example on securing that the state of the system should always be better after a migration is shown for two nodes. First in figure 5.4 (a) we look upon the nodes before any migration is performed. It can be seen that node A is below the lower threshold and thus it is overloaded. Therefore it could be an advantage to move a process from node A to node B. But if the result of the move is that node B falls below the lower threshold as shown in figure 5.4 (b) then the migration is not preferable as the system has not gained anything. But if the result of the migration is as seen in figure 5.4 (c) then the overall distance in available resources from the system average available resources has become smaller and thus the system has gained from the migration.

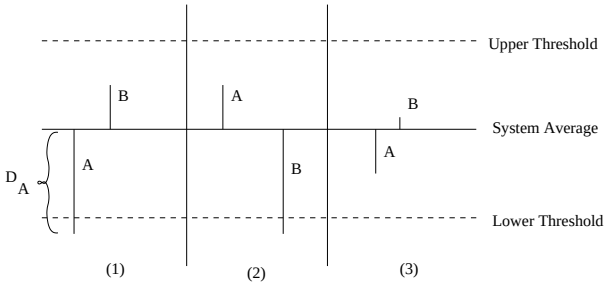


Figure 5.4: An illustration of the resource rule.

This approach is used in order to avoid thrashing. If the state after a migration has been performed can be predicted before the migration is actually carried out the location policy can base its choice of node on that prediction. If the prediction shows that the state of the system is not better after a process has been migrated to a certain node, another node can be considered as an alternative target or migration can be canceled if no node can be found that gives a better overall result after migration.

Generally the following calculations must hold in order to migrate a process from one node to another.

First we calculate the average available resources after the migration has been performed for each of the nodes.

$$R_A^* = \frac{R_A \cdot NoProc_A}{NoProc_A - 1}, \tag{5.3}$$

$$R_B^* = \frac{R_B \cdot NoProc_B}{NoProc_B + 1}, \tag{5.4}$$

where R_A^* and R_B^* are the predicted resources available on the nodes after migration is performed and R_A and R_B are the available resources before migration is performed. $NoProc_A$ and $NoProc_B$ are the number of FROST processes on nodes A and B respectively.

Then we see whether the system gains something by allowing migration. First we calculate the absolute distance, D, of the resources on a node from the system average. This is done for both nodes using both known resource values before migration, R, and predicted resource values after migration, R^*. An illustration of these distances are shown in figure 5.4 showing the distance, D_A, for node A.

$$D = |R - S_{Avg}|, \tag{5.5}$$

where S_{Avg} is the system average.

Then if the combined distances after the migration is smaller than the distances before the migration the system will proceed with the migration.

$$D_A^* + D_B^* < D_A + D_B, \tag{5.6}$$

where D_A^* and D_B^* are the distances after and D_A and D_B are the distances before migration.

If this holds and R_A^* is above the lower threshold the node is chosen as a target for migration. Otherwise a new node will be chosen and the process will be started over or migration will be canceled and the process restarted on its current node.

After a node has been chosen as a target for the migration we need to ensure that the load on the chosen node has not changed for the worse. This can be done by letting the sending node request the load of the node to which it will send a process just before actually sending the process. Then if the load has changed to the worse the process could be withheld and another node could be chosen as target for migration.

5.4.2 Locating a Node

In order to be able to use the prediction described above the location policy first has to choose a node among all the nodes in the system. Depending on whether the state of the node on which the location policy is running is receiver or sender initiated two approaches must be considered. These are shown in figure 5.5

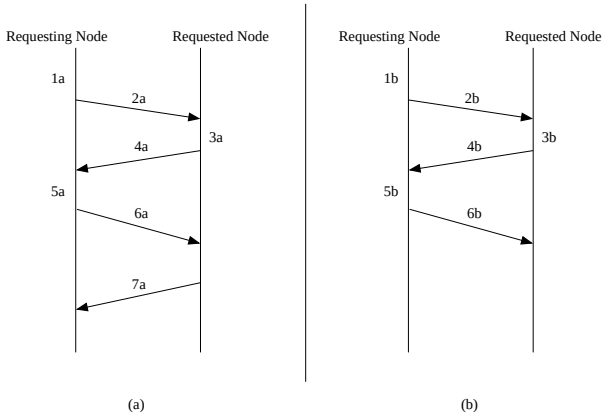


Figure 5.5: Locating a target node. (a) shows the receiver initiated approach and (b) shows the sender initiated approach.

In figure 5.5 (a) the receiver initiated approach is shown. Below is the protocol for the receiver initiated approach.

It should be noted that due to the structure of the FROST system only a single task of each assignment on a node is calculated at the same time. Therefore we do not allow a process to be migrated to a node that already executes a task from the same assignment as it would have to wait until the already present task had finished before it could continue calculating.

1a First a node discovers that the current system state is receiver initiated and that it is underloaded and it therefore finds a node that is more loaded than itself.

2a It then issues a request to that node for a process and with the request it states the maximum amount of memory that process may use.

3a The requested node first checks whether it is already performing a migration and if so it rejects the request. Otherwise it locates a process which complies with the demands.

4a The requested node replies with the current amount of available resources, the number of currently running processes and the id of the chosen process.

5a If the requesting node already has a process from the same assignment we are not interested in receiving it, it is rejected and another is requested. If the requesting node does not already have the process, the prediction described in section 5.4.1 is performed using the new resource information from the requested node.

6a If the prediction shows that it would be advantageous to move the process, an OK is sent to the requested node.

7a When an OK is received the process is migrated.

In figure 5.5 (b) the sender initiated approach is shown and we state the protocol for this approach below.

1b The node discovers that the system state is sender initiated and that it is overloaded and it therefore locates a node which is not overloaded. It then finds the best process for migration.

2b It then issues a request for migration to the other node sending along the id of the chosen process, and the memory usage of the process.

3b The requested node checks whether it is already performing a migration and if so it rejects the request. Otherwise it checks the id of the process and the memory usage. If the id already exists on the requested node, the request is rejected. If the memory usage of the process is unacceptable the request is rejected. Otherwise it is accepted.

4b The requested node returns a message containing its currently available resources and the number of running processes if the request is accepted, else a rejection is returned.

5b The requesting node performs the prediction procedure described in section 5.4.1 using the new resource information and process number from the requested node.

6b If the prediction shows that it would be advantageous to move the process, the process is migrated.

As can be seen in the above protocols we send the newest available resource number and the number of currently running processes every time we go through the protocols. These information could also be sent every time the available resources are updated as part of the information policy. We choose not to send the number of processes regularly as there is only need for it in the above protocols. The updated resource information is sent in order to get the latest information for the prediction procedure.

From the protocols, points 5a and 3b respectively, it can be seen that the FROST system should not allow a process to migrate to a node which is already calculating a process from the same assignment. This is due to the way FROST handles assignments. In FROST only a single work unit per assignment is calculated on each node at any given time. If a checkpoint was to arrive on a node which was already running a process from the same assignment the checkpoint would have to wait until the other process had finished before it could proceed. Therefore it is more advantageous for a process to stay at a heavily loaded node and perform calculations, even though it may not be much, than it would be to migrate and perform no calculations.

CHAPTER 6

Checkpointing

In section 3.5 some technical demands were specified which must be considered during the design phase. In this section we will design the facilities needed for extracting the state of the process and saving it to disk and to be able to recover the process again. This id done before the design of the preprocessor in order to point out the elements that need to be handled by it.

In order to extract the process state there are a number of different issues that must be considered. We have identified the following:

Processor state: The state of the processor with regard to the particular process. This state consists of the runtime stack, program counter and the registers, and must be captured in order to restart the process on another machine.

Process variables: In order to transfer a process to another machine, it is necessary that all variables used in the process are transferred.

Data marshaling: When transferring variables in a heterogeneous environment it is necessary to make sure that they are interpreted the same way on all architectures.

A general design issue to the checkpointing procedure is that when performing a new checkpoint, the last checkpoint made is not removed before all data is safely checkpointed. This is necessary in order to be sure not to loose any data if the machine breaks down during the checkpoint procedure.

6.1 Processor State

The processor state information consists of the contents of processor registers [Sta98]. This information determines where in the code the process is executing and how further execution will continue. It needs to be transferred with the migrating process, in order to restart the execution on the destination machine. The information includes user registers and control and status registers such as the program counter and a stack pointer. Furthermore it is necessary to transfer the runtime stack itself which holds return addresses and parameter values for method invocations etc. The processor state information is, as the name indicates, very processor dependent and therefore it is very important to consider heterogeneity when transferring this information. This is done in the following sections.

The process control information [Sta98] is less important when migrating processes in FROST. Much of the information is dependent on the actual execution on the machine where the process is executing, and will be automatically generated when the process is restarted. Whether the process is in virtual memory or not depends on the operating system on which it is executing[1]. There are, however, elements that are both dependent on the operating system and the executing process, such as I/O and file access, but access to this kind of resources has been prohibited in FROST for security reasons [GK02], and will therefore not be considered any further.

6.1.1 Runtime Stack

As we wish to perform process migration in a heterogeneous environment, there are some demands to the transferral of the runtime stack. In a heterogeneous environment it is necessary to translate the stack between the different architectures, e.g. by using an intermediate format known to all architectures. As the runtime stack can be, and possibly is, constructed very differently on the different architectures, we wish to make a completely independent way of extracting the stack.

[1] We cannot control this as we are designing for user space migration.

We have adopted Chanchio and Sun's [CS96] way of keeping track of function calls in MpPVM. For this purpose they use a *control stack* to which a label is pushed when entering a function and popped when leaving the function. When migrating a process, the control stack is transferred and used to build the runtime stack by performing the actual function calls. The runtime stack is rebuilt by jumping down to the function call, performing the function call and jumping to the next function call or to where the execution of the process must continue.

The example code in figure 6.1 shows an example on the functionality that is needed in order to keep track of the runtime stack using a control stack. Furthermore it shows the code necessary to recover from a checkpoint.

```
int main() {
  if(ExecutionMode == RECOVER){
    RecoverControlStack()
    switch(NextLabel()){
    case 1:
      goto _FirstLabel;
      break;
    }
  }

  FirstMethod();

_FirstLabel:
  Push(1);
  int result = SecondMethod();
  Pop();
  .
  .
}

void FirstMethod(){
  // No checkpoint made in
  // this method
}
```

```
int SecondMethod(){
  if(ExecutionMode == RECOVER){
    switch(NextLabel()){
    case 2:
      goto _SecondLabel;
      break;
    }
  }

  // Variable assignments etc.

  Push(2);
  CheckpointControlStack()
  CheckpointVariables();
  Pop();
_SecondLabel:
  if(ExecutionMode == RECOVER){
    RecoverFromCheckpoint();
    ExecutionMode = NORMAL;
  }
  .
  .
}
```

Figure 6.1: A simple example on how the control stack is used. Only methods containing a checkpoint are pushed to the stack. Since FirstMethod does not lead to a checkpoint it does not need to be considered when performing or recovering from the checkpoint.

In order to handle this functionality, two data structures are needed: the control stack that holds the labels of the method-calls and a flag that specifies the execution mode of the process. The state of the execution mode is used to determine whether the process is running normally or recovering from a checkpoint. During normal execution, most of the control stack code is not executed. Only the Push and Pop methods and the CheckpointVariables method are called in order to perform the checkpoint. In the RECOVER-state a simple procedure is followed in order to rebuild the runtime stack from the control stack. The labels recovered from a checkpoint file are run through using the call to NextLabel. These labels are used to enter the correct method, where the checkpoint was performed as described above.

The Push and Pop methods are placed on each side of the method-calls to all methods that leads to a checkpoint. We define a method that leads to a checkpoint, as a method that has not returned before a checkpoint is performed. Hence, FirstMethod in figure 6.1 does not lead to a checkpoint but SecondMethod does. If for instance SecondMethod was invoked from inside FirstMethod, FirstMethod would also lead to a checkpoint.

When a checkpoint is performed in SecondMethod, the stack will hold the values 1 and 2. The control stack is also included in the checkpoint and the values are used during recovery. Figure 6.2 shows how the recovery code builds the runtime stack using the recovered labels.

After the execution mode is set to normal, the process is recovered and the execution continues. In order for the checkpoint to make sense in the example, the main-method must somehow be dependent on the executed statements in SecondMethod, and there should be more statements after the

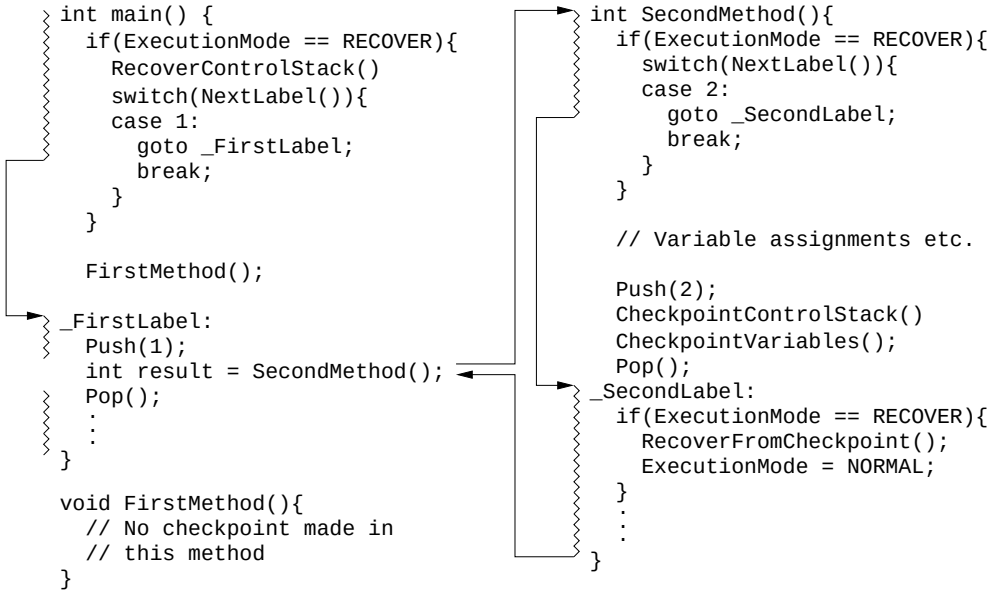


Figure 6.2: The same example as in figure 6.1, showing the thread of execution during recovery. Normal execution proceeds after the execution mode is set to normal last in SecondMethod.

checkpoint.

The control stack will be implemented as a class in order to support the extra functionality needed, compared to a standard stack. This functionality is basically the ability to checkpoint and recover the stack itself. By placing this functionality here it is easy to implement the NextLabel-method used when the process is in the recovery state. The class diagram for the ControlStack class can be seen in figure 6.3.

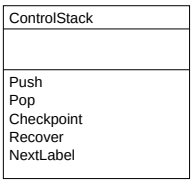


Figure 6.3: The class diagram for the class ControlStack.

The Checkpoint and Recover methods respectively saves the stack to and loads the stack from persistent media. It must be saved in a machine independent manner, as described in a later section, in order to make sense when the process is transferred to a machine with a different architecture. Whether the stack should be rebuilt or not during recovery depends on where the labels are inserted in the source code. In figure 6.2 the labels are placed before the call to Push, which means that the stack will be rebuilt at runtime. If the call to Recover rebuilds the control stack, the labels must be placed after the call to Push at regular method invocations and before the call to Pop at the checkpoint. After the Recovery method has been invoked, the NextLabel method will return the next label to jump to.

When the last jump has been performed (the jump to _SecondLabel in figure 6.2), the control stack and hence the runtime stack has been recovered. All that is needed now is to recover the user variables, which is explained further in a later section.

6.1.2 Program Counter and Registers

The way that we handle the runtime stack makes it extremely easy to handle the program counter and the registers. The program counter is set automatically when recovering, as the actual method invocations are carried out, and a jump to the correct source line is performed at the end. Our checkpoints are always performed between source lines and not in the middle of a calculation, and therefore also the registers are handled automatically.

6.2 Process Variables

In order to migrate a process it is necessary to transfer all variables accessed by that process. In FROST this can be limited to variables and objects accessible to the CalculationCode object (see figure 6.4) and the calculating thread. This can furthermore be limited as some variables are initialized automatically when the calculation code is loaded. This feature is explained further in the following section. Afterwards handling of the variables accessible to the calculating thread is explained.

6.2.1 Member Variables

The member variables are first of all the variables in the base classes CalculationCode, Data- and ResultObject and Data- and ResultLump[2]. These classes are defined in [GK02]. The Data- and ResultLump classes represent the smallest entity of either the data or the result of a work unit respectively. Also variables added to the inherited classes by the user are considered as member variables and must be transferred during migration. The class diagram of the entire calculation code component can be seen in figure 6.4, which is taken from [GK02].

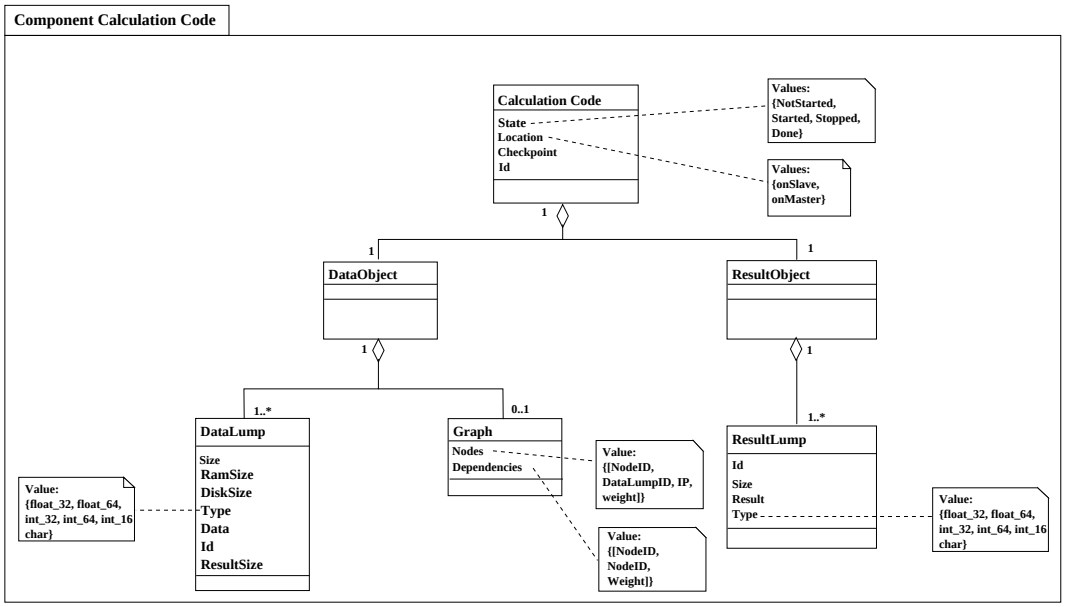


Figure 6.4: Class diagram of the calculation code component. The figure is taken from [GK02].

The CalculationCode, DataObject and ResultObject classes must be handled specially as these are expected to already be loaded and initialized on the destination machine when a process is migrating[3]. This means that when recovering from a checkpoint, we do not need to allocate and load these objects,

[2]The Graph class is not necessary to checkpoint as it is only used on the master and we only migrate processes between slaves.

[3]If the calculation code is not loaded on the destination machine before migration, it must be loaded before the checkpoint is recovered.

but only set some of the member variables correctly.

The Id and Location variables in the CalculationCode class (see figure 6.4) is both set upon initialization of the class and does not need to be transferred upon migration. The Id variable is even unique for the local machine that the calculation code is residing on, and therefore it makes no sense to transfer it to another machine. The State variable is set during runtime, and as we rebuild the runtime stack by performing all the method invocations, this variable will be set correctly without any further handling. The Checkpoint variable indicate the ID of the last checkpoint made. This ID must be set by the slave component in order to inform the CalculationCode class about which checkpoint to recover from. The CalculationCode controls the access to the DataObject and ResultObject classes and must therefore take care of saving them.

The DataObject and ResultObject classes does not contain any variables except for one or more members of the DataLump and ResultLump classes respectively. They contain a number of simple variables[4] which must all be saved. In order to ease the saving of variables a special class will be made.

The DataStream Class

The DataStream class provides a simple interface to the saving of data. Furthermore it provides the functionality necessary to transfer the saved data in a heterogeneous environment. This feature and the saving of pointers and more advanced structures such as arrays and STL-containers[5] are described in a later section regarding data marshaling. This section concentrates on the DataStream class interface.

The interface must include facilities to save and load data. This require a checkpoint and a recover method that can handle several types of data. For simple types, this is not a problem but for user-defined types special care must be taken.

In order to handle user-defined objects in a architecture independent manner it is necessary to access member variables through their variable names. A possibly non-portable method would be to access member variables through offsets from the object address, but that would require that the compiler lays out the variables in a deterministic way which makes it a non-viable solution. To be able to access private variables through variable names it is necessary to access them from inside the class. This requires a public method in the class that handles saving of all member variables. When such an object is encountered, the checkpointing or recovering method must recognize it as a user-defined object and invoke the saving or loading method in the object.

Due to the way we have chosen to checkpoint objects we limit ourselves from checkpointing third party libraries. It is necessary for us to be able to insert the saving and loading methods in the source code in order for us to checkpoint an object, and that is not possible with third party libraries[6]. If third party libraries are used, it is required that they are initialized in the constructor of the calculation code.

The use of variable names when checkpointing sets some limitations on the types of data. Variables declared as constants and references must be instantiated at their declaration, as the compiler will not allow us to initialize them later.

As there naturally will be several variables in each checkpoint, it is necessary to have delimiters around the checkpointing and recovering. These delimiters can consist of a Start and an End method that initializes and finalizes the checkpoint respectively. These methods and the Checkpoint and Recover methods forms the interface of the DataStream.

6.2.2 Runtime Variables

When a calculation is running it is most likely that some variables are created on the stack, or even on the heap, in the methods. These variables are referred to as runtime variables, and must also be transferred during migration as they are a part of the process state. This can be done in several ways,

[4] Simple variables are variables of simple data types.
[5] `http://www.cppreference.com/cpp_stl.html`
[6] It is, however, possible with open source libraries but will require re-compilation of the libraries.

where some are more suitable to be used in a heterogeneous environment than others.

When performing a checkpoint, it is necessary that we capture the state of all variables at the same time. We cannot expect all variables to be in scope, when the checkpoint is performed, and therefore special care has to be taken. To illustrate this, we have set up a simple example:

A-METHOD()

1: $x \leftarrow 0,\ y \leftarrow 0$
2: ...
3: ANOTHERMETHOD(x)
4: ...
5: **return** x

ANOTHERMETHOD(`var` x)

1: **for** $i = 0$ to 10 **do**
2: $x \leftarrow x + i$
3: CHECKPOINTVARIABLES()
4: **end for**

The y variable is not in the scope of ANOTHERMETHOD where the checkpoint is performed. As we are making checkpoints instead of migration points we have to save data each time a checkpoint occurs, and not just when the actual migration is carried out. Hence, we cannot save variables that are in the scope of the current function call, return to the previous call and save variables available in that scope and so on. When the checkpoint has been performed, the execution must proceed. In order to handle this problem, it is possible to analyze the stack backwards through the function calls without returning from them, and then jump back to the original address and continue execution. This method is difficult to handle and inappropriate in a heterogeneous environment as analyzing the stack is not very portable. The structure of the stack depends both on the compiler used and the machine architecture the program is compiled for. Furthermore the locations of the variables cannot be deterministically determined if compiler-optimizations are switched on, unless the algorithms used by the compiler are known.

Another solution would be to save all variables in the current scope that are not transferred to the next scope. Hence, the variable y above is checkpointed before the invocation of ANOTHERMETHOD. This will, however, require that we keep track of all scopes in the checkpoint. It also adds to the complexity when we need to keep a backup of the previous checkpoint in the current scope in order not to overwrite the last checkpoint.

A simple solution can be found if it is possible to checkpoint all variables at the same time. Hence, we need a method where we can access the variables at their correct location, without them being in the scope of where the checkpoint is performed. Furthermore the process must continue from the same point after the checkpoint has been performed. In order to access the correct location of the variables, we need to find the address at runtime. This can be done by keeping an array of addresses of the variables declared in each scope. This solution is somewhat similar to a *display* in the compiler terminology where it is used to keep track of nested scopes [App98]. The following is a simple example on how the addresses can be saved, where $DataStack$ is the global array containing the addresses:

SOMEMETHOD()

1: $int\ a$
2: $float\ b$
3: $DataStack[0] \leftarrow address\ of\ a$
4: $DataStack[1] \leftarrow address\ of\ b$
5: ...

When making the checkpoint the addresses must be casted to the correct type in order for the DataStream class to recognize it and save it correctly.

This way of saving runtime variables gives us the possibility of saving variables that are not in the current scope without analyzing the runtime stack. It is a portable solution as the addresses determined at runtime are used correctly and with the correct data type.

This solution has a drawback, though. It is necessary that the number of variables that need to be saved can be determined at compile time, which limits the code from using recursive method calls[7]. They can, however, be used between checkpoints, but there cannot be performed a checkpoint inside a recursive call. In order to handle this situation it is necessary to use the same procedure as with the runtime stack. Hence, it must be possible to push variable addresses to the data stack at runtime. This makes it impossible to determine the type of a variable at run time, and therefore it is necessary also to push the type of the variable. For simple data types, this is a simple demand, but if user-defined types must be supported, it introduces a problem. The code that saves the variables from the data stack must be defined at compile time where the user-defined types must be added to the mapping between the value pushed onto the data stack and the data type.

To support checkpointing in recursive method calls, the data stack is designed as a class somewhat similar to the control stack. There must be methods to push variable addresses and the variable types. It is, however, not necessary to be able to pop values one by one. When a method returns, all variables declared (and pushed) in that method, can be popped altogether. Hence, if a special method-delimiter is pushed before the variable declarations they can be popped all at once when the method returns.

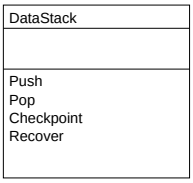


Figure 6.5: The class diagram for the class DataStack.

It is most natural to place the code for checkpointing and recovering variables in the DataStack class and then adjust the code at compile time to include user-defined types. This gives a class as shown in figure 6.5 with an interface very similar to the control stack.

The location of the call to Checkpoint is already determined due to the functionality of the DataStack class. All variables can be checkpointed from any scope, and therefore the call to Checkpoint can be placed where the checkpoint is to be performed. The location of the call to Recover, however, must be considered very carefully. This is depicted in the following example:

1: $MyClass\ x$ {Declaration of x of type MyClass}
2: ...
3: x.METHODINMYCLASS()
4: ...

If a checkpoint is performed inside METHODINMYCLASS it is necessary that x is recovered before the call to METHODINMYCLASS. To ensure this, variables must be recovered in the scope they are declared. This will require that a call to Recover only recovers variables belonging to a single scope. As noted earlier, scopes are separated by the method delimiters.

In the following section we will discuss how more advanced data structures are handled.

6.3 Data Marshaling

As with the control stack, the transferral of process variables must be done in an architecture independent format when migrating in a heterogeneous environment. In order to ensure this, we have chosen to represent the checkpoint data using the External Data Representation (XDR) standard [Sri95]. The XDR standard specifies how all the simple data types must be encoded, and can therefore be used independently of the architectures on which data is saved and loaded.

The saving of pointers must be handled similarly to the saving of simple data types, by saving the data that is pointed to. There must, however, be taken special care of pointer aliases[8]. It is very important

[7] When we use the term recursive method calls we also refer to mutual recursive method calls.
[8] Pointers that points to the same address.

that they are recognized during the checkpointing and saved correctly and, even more important, they must be loaded correctly in order to ensure that the calculation code executes correctly. Loaded correctly means that two pointers, pointing to the same address when the checkpoint is saved, must also point to the same address when the process has been recovered.

Array sizes must also be recognized in order for all values to be saved. In C/C++ it is not possible to tell the difference between pointers to a single element or an array, and it is therefore necessary to handle arrays specially. This can be done by logging the allocation of arrays, and keeping a table of arrays and their sizes. This can either be done by locating all memory allocations in the source code and insert code that builds such a table from the address and number of elements determined at runtime, or the **new** operator can be overloaded. In order to simplify the code that must be inserted to support migration, we choose the latter.

CHAPTER 7

Preprocessor

As we have chosen an indirect way of extracting the state of our migrating processes it is necessary to insert checkpoint code into the users source code as described in chapter 6. In order for users to be able to exploit the process migration feature, a great deal of transparency is needed, as correct insertion of checkpoint code can be rather complex. In order to achieve the transparency needed we have chosen to develop a preprocessor to handle the analyzing and modification of the user source code.

The work of the preprocessor is mainly to insert the use of the data structures designed in chapter 6, which requires careful analysis of the source code. The design of the preprocessor is carried out in the following sections. The aim of the preprocessor is to obtain an optimal interaction with the user, in such a way that the user experiences maximum transparency contemporary with the preprocessor generating the most optimal code.

In the following section we will start out by introducing the general structure of the preprocessor. Afterwards we will describe some user requirements we have set in order to achieve a more optimal preprocessor and to limit the complexity of it. Finally the different parts of the preprocessor will be designed including the intermediate format.

7.1 General Structure

The general structure of the preprocessor can be seen in figure 7.1. We have chosen to divide the preprocessor into three major modules. The lexer and the parser is depicted as one module, as these are automatically generated using a compiler-generator tool.

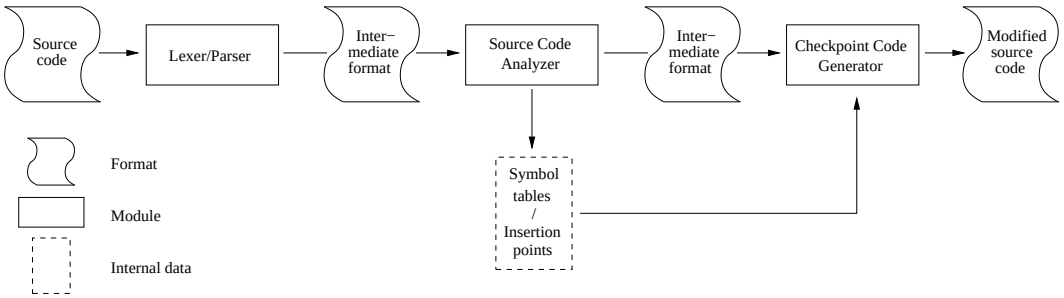


Figure 7.1: The overall structure of the preprocessor.

The last two modules is the main modules in the preprocessor. The Source Code Analyzer analyzes the code in order to determine the structure and locate the needed symbols and places to insert checkpoint code. This information is handed over to the Checkpoint Code Generator which prepares and inserts the checkpoint code into the user source code. The output is the modified source code, which can be compiled with a regular compiler.

7.2 User Requirements

We have chosen to make some requirements to the user both in order to simplify the complexity of the preprocessor but also to make it possible for the user to optimize the code generated by the preprocessor. These requirements consist of making the user insert certain tags in the source code,

which is used by the preprocessor. It is possible to make a preprocessor that do not require any user interaction as is done in the MpPVM system [CS96]. We do, however, believe that we can obtain more optimal results with some user interaction as the user often has knowledge of his source code which is difficult to capture and implement in a general way in a preprocessor.

7.2.1 Marking the Checkpoint Location

In the MpPVM preprocessor migration points are inserted automatically based on data analysis of the user source code [CS96]. The location of migration points is not as important as the location of checkpoints[1], as the migration points do not induce as much overhead as checkpoints. Checkpoints require extraction and saving of the process state each time they occur, and therefore the location is very important. The placement of the checkpoint location is a balance between how much data that may be lost during migration or machine failure, and how much overhead there is induced by the checkpoint.

We believe that the users of FROST has great, or at least some, knowledge of the algorithms they are implementing, and therefore they are better suited to place the checkpoints optimally compared to a preprocessor. We have therefore chosen to let the user specify where checkpoints should be performed in the source code.

The tags that the user must insert into his calculation code must be well-defined in order for the preprocessor to recognize them. We have chosen the following syntax for the location of checkpoints:

```
// −−FROST−− CHECKPOINT
```

Hence, the tags are inserted as comments in the original source code.

It must be possible to insert several checkpoints as it cannot be expected that all calculations is performed in a single loop-structure. Furthermore, we will make it possible to place checkpoints in all user-defined methods, as long the complete source code is available to the preprocessor.

7.2.2 Marking Variables

We will also make some requirements regarding variable declarations. One requirement is that the user must indicate the variables which should not be in the checkpoint. There are two reasons for this requirement. We have limited ourselves from checkpointing third party libraries in section 6.2.1, and in order to simplify the preprocessor by not requiring it to locate library variables, we need the user to mark the variables which are not supposed to be checkpointed. The other reason is, that by permitting the user to indicate variables which are not to be checkpointed he has the possibility to optimize the checkpointing process. By indicating the variables that are initialized in the constructor of an object and not assigned later to be excluded from the checkpoint, the checkpointing process can be optimized.

As with the checkpoint location we have defined tags to be inserted around variables which are not to be checkpointed:

```
// −−FROST−− DO_NOT_CHECKPOINT START
 ... ( declarations of variables that shall not be in the checkpoint)
// −−FROST−− DO_NOT_CHECKPOINT END
```

Variables declared inside these tags will not be considered when inserting the checkpoint code into the calculation code.

For convenience we require that all stack variables are declared in the beginning of a method. When we are rebuilding the runtime stack as described in section 6.1.1 we use **goto**. Because some compilers do not allow jumps across variable declarations, it is necessary that variables are declared in the beginning of methods. We could make the preprocessor move the variable declarations by itself, but for simplicity we require the user to take care of it.

All of the user requirements can be handled automatically by the preprocessor, and we believe they

[1] By the location of checkpoints is meant where the checkpoint is performed in the source code.

should be handled in a final version, but we make these limitations for now. We believe that it should always be possible for the user to specify the location of checkpoints.

7.3 The Parser

In this section we will describe the statements that must be recognized in the source code. This can be used to design the grammar which is used to generate the parser by the compiler-generator tool. Furthermore the following sections will aid the design of the Source Code Analyzer.

7.3.1 FROST Tags

It is obvious that the preprocessor needs to be able to identify the FROST tags which were defined in section 7.2.1 and 7.2.2.

The marking of variables that shall not be checkpointed is used to aid the analyzer in the identification of variable declarations. If these tags were not required, it would be necessary to go through all the source code available to the preprocessor and identify the variable types which is not defined in the available source code.

The marking of the checkpoint locations is very important to the Source Code Analyzer. As it determines where the checkpoints must be performed, it makes the foundation for most of the code analysis. This is explained further in the following section regarding method invocations.

7.3.2 Method Invocations

The locating of method invocations is very dependent on the checkpoint location. It is only necessary to locate method invocation which leads to a checkpoint as described in section 6.1.1.

The locating of method invocations is used to place Push and Pop calls around the invocations as described in section 6.1.1, and it is not necessary to trace the stack into methods where a checkpoint is never performed. This can be seen in the example code shown in figure 6.1 where only SecondMethod leads to a checkpoint, and therefore the Push and Pop calls are only placed around the invocation of this method.

The parser will locate all method invocations, and then it is the job of the Source Code Analyzer to determine which methods the Push and Pop calls shall be placed around.

7.3.3 Variable Declarations

In order to insert the checkpointing code it is necessary to locate all variables and their types that exists in the scope of the checkpoint location. This includes both all user-defined classes and variables declared in methods that leads to a checkpoint. The information shall be used to extend the Checkpoint and Recover methods as described in section 6.2.2, so that it can handle user-defined types. Furthermore it is needed when the saving and loading methods are to be inserted in the user-defined classes as described in section 6.2.1. Finally, it is necessary to push all addresses and types of variables, which is declared in methods that leads to a checkpoint, to the data stack when they are declared.

All the recognized statements described in the above section is processed by the Source Code Analyzer in order to determine where extra code must be inserted. Furthermore, elements needed for the code generation are found. The code that is to be inserted is considered in the following section which leads to the design of the Source Code Analyzer.

7.4 The Checkpoint Code Generator

The Checkpoint Code Generator is used to insert the code that performs the checkpoint. The code that needs to be inserted was described in chapter 6. We have chosen to design the Checkpoint Code

Generator before the Source Code Analyzer in order to determine which information that is needed from the analyzer to be able to insert the migration code.

The code that needs to be inserted consists in the following:

- Logging of variable addresses and types
- Checkpoint and recover methods for user-defined classes
- Push and Pop around method calls
- Checkpointing data
- Recovery code

In the following we will describe in more detail the code that is to be inserted and the location of the insertion, which will lead to the information needed from the code analysis.

7.4.1 Logging of Variable Addresses and Types

The logging of variable addresses and types must be done in order to checkpoint stack variables in the calculating methods. Hence, this section only applies to variables declared in methods, and as noted above it is only necessary to log variables in methods that leads to a checkpoint.

The logging is done using the DataStack class which was designed in section 6.2.2. To log a variable, the address and its type must be pushed to the data stack. Hence, calls to push must be inserted for all variables declared in each method leading to a checkpoint. In order to ease the popping of variable addresses when the method returns, a method delimiter must be pushed to the data stack. In this way all values can be popped at once.

As variable types cannot be pushed directly to the data stack, we need some sort of mapping from variable types to integers. In this way we can push the variable address and an integer denoting the variable type. As user-defined variable types is not known on beforehand, it is necessary that the Source Code Analyzer provides us with this information. Hence, we need two tables, a variable table and a mapping table holding a mapping from variable types to integers.

Content	Variable name	Variable type	Location
Type	Text	Number	Line number

Table 7.1: The variable table holding variable names and their types required when logging stack variables.

The mapping table is depicted in table 7.1. As variables are in scope when the logging is performed, the variable names can be used directly to retrieve the address. The variable types have been assigned a number, which represents a variable type name as depicted in table 7.2. The location of where to insert the logging must be available, e.g. as a line number in the source code.

Content	Variable type name	Type value
Type	Text	Number

Table 7.2: The mapping table that maps the actual variable name to the type value assigned to it.

As the Checkpoint method in the DataStack class is used to checkpoint all stack variables it must know all possible variable types. As described in section 6.2.2, this is not possible before the source code is run through the preprocessor, and therefore it is necessary that the preprocessor adds handling of any user-defined types to the DataStack class. The mapping table in table 7.2 must be used for this purpose.

All variables is required to be declared in the beginning of method calls. The logging of variables must be done both during normal execution and during recovery, as normal execution always proceeds when recovery has finished. Therefore the location of variable logging must be before the recovery code jumps to a method invocation or checkpoint. Thus, we will insert logging code just after the variables have been declared.

7.4.2 Checkpoint and Recover Methods in User-defined Classes

The insertion of checkpoint and recover methods in user-defined classes is very simple. The only requirement is that they are inserted as public methods as they are invoked from outside the class. The class table only has to provide the location of the insertion and the names of the variables, as, of course, the variable types are known from inside the class.

Content	Variable name	Location
Type	Text	Line number

Table 7.3: The class table holding information needed for inserting checkpoint and recover methods in the user-defined classes.

The class table is depicted in table 7.3, where variables in the same user-defined class can be given the same location.

7.4.3 Push and Pop Placement

The pushing and popping of values to and from the control stack has already been defined in section 6.1.1. A label must be pushed before a method is invoked and popped again when the method returns. As noted above, we only need to keep track of methods that leads to a checkpoint, and therefore the Source Code Analyzer must provide information about which methods that leads to a checkpoint. This can possibly consist in the direct insertion point of the methods, as indicated in table 7.4. The insertion of push and pop calls also includes the insertion of labels that can be used to jump to, when the recovery code is rebuilding the runtime stack. The labels are also included in the table, but whether it is filled in by the Source Code Analyzer or the Checkpoint Code Generator does not matter, as long as both the labels and the label values are unique.

Content	Location	Label	Label value
Type	Line number	Text	Number

Table 7.4: The method table holding information needed for inserting push and pop calls around method invocations that leads to a checkpoint.

The label and the label value is used by the recover code described in a later section.

7.4.4 Checkpointing Data

The checkpointing of data consists in activating the Checkpoint methods in all the different classes. This includes the CalculationCode class, the ControlStack class, and the DataStack class. Furthermore the Start and End methods from the DataStream class must be invoked in order to initiate and finalize the checkpoint. All this code must be inserted where the checkpoint is to be performed. The code that is to be inserted is not dependent on the user source code, and thus all that needs to be provided is the location of the checkpoint. This information is very similar to the information needed for the push and pop placement described in table 7.4, as labels at checkpoints are also needed when generating recovery code. By adding a column to the method table in table 7.4 indicating whether the entry is a method or a checkpoint, the table can hold information about checkpoints too.

7.4.5 Recovery Code

The recovery code consists of the code for rebuilding the runtime stack and the code for loading variables. The location of these two elements are very dependent on each other. The code for rebuilding the runtime stack consists of the code that checks a label from the control stack and uses it to jump to the correct method call, as depicted in figure 6.1. This code must be placed after the logging of variable addresses and types and hence, after variable declarations. This is due to two issues. First of all, we cannot jump across variable declarations as described in section 7.2.2. Secondly, we also need to log the variable addresses during recovery, as checkpointing will be performed again, after the normal execution resumes.

The code that loads the stack variables must be placed in the scope of the variable declarations. This is in fact not a requirement due to the functionality of the DataStack class, but as described in section 6.2.2, it is possible that a method is invoked on a stack variable, before the checkpoint location is reached. Hence, the recovering of variables must be done after the variable declarations and before a method is invoked, and this can either be before a jump to a method is performed, or between the label that is jumped to and the method invocation. The latter requires that variable recovery code is added at each label in the current method and therefore the former is chosen. This situation requires that the Recover method of the DataStack class only recovers the most recent scope of variables. The method delimiters on the data stack can be used for this purpose.

No symbols is needed for the insertion of these two code elements, but the Source Code Analyzer must provide information regarding the location of where the code is to be inserted. This information can be placed in the method table by adding an extra column. The complete table is shown in table 7.5

Content	Location	Label	Label value	Type	Recover location
Type	Line number	Text	Number	Method/Checkpoint	Line number

Table 7.5: The complete method table holding all information needed regarding insertion of push and pop calls, insertion of code for checkpointing data, and insertion of recovery code.

The code used for initiating the checkpoint and loading the calculation code member variables[2] can be performed in the CalculationCode class itself. Therefore only call to End in the DataStream and the code for setting the execution mode must be inserted at the point of the checkpoint.

7.4.6 Algorithm for the Checkpoint Code Generator

From the information found in sections 7.4.1 through 7.4.5 we can design the algorithm needed for the Checkpoint Code Generator. In algorithm 6 it is implied that only scopes and method invocations leading to a checkpoint is considered.

7.5 Source Code Analyzer

The Source Code Analyzer[3] must analyze the user code, in order to locate the information needed to fill in the tables described above. When filling the variable, mapping and method tables it must be known which methods that leads to a checkpoint, as it is only necessary to consider these methods. The methods can be found using the fix point algorithm shown in algorithm 7. The $InvokesMethods$ data structure is an array of sets, where each set holds the methods that are invoked by the method which the entry belongs to:

[2] The calculation code object is already instantiated, but some member variables must be set correctly.
[3] It is called a Source Code Analyzer even though it operates on the intermediate format.

Algorithm 6 Algorithm for inserting the checkpoint code.

INSERTCHECKPOINTCODE()

```
1:  for all entries in the mapping table do
2:      insert handling into DataStack class
3:  end for
4:  for all entries in the SymbolTable do
5:      insert logging code at Location
6:  end for
7:  for all distinct locations in the class table do
8:      insert checkpoint and recover methods where
9:      all variables with the same location are
10:     checkpointed or recovered in the same method
11: end for
12: for all entries in the method table do
13:     if Type = Method then
14:         insert push, pop and label
15:     else
16:         insert checkpoint code
17:     end if
18: end for
19: for all distinct recover locations do
20:     insert recover code
21: end for
```

Algorithm 7 Algorithm for finding methods that leads to a checkpoint.

FIND-METHODS()

```
1:  LeadsToCheckpoint ← {}
2:  for all methods A do
3:      if A includes checkpoint then
4:          LeadsToCheckpoint ← LeadsToCheckpoint ∪ {A}
5:      end if
6:      InvokesMethods[A] ← {}
7:      for all method invocations B in A do
8:          InvokesMethods[A] ← InvokesMethods[A] ∪ {B}
9:      end for
10: end for
11: repeat
12:     for all entries A in InvokesMethods do
13:         if (InvokesMethods[A] ∩ LeadsToCheckpoint) ≠ ∅ then
14:             LeadsToCheckpoint ← LeadsToCheckpoint ∪ {A}
15:         end if
16:     end for
17: until LeadsToCheckpoint does not change
```

When the algorithm has finished, $LeadsToCheckpoint$ holds the names of all methods leading to a checkpoint.

7.5.1 The Variable and Mapping Tables

The variable and mapping tables are easily filled using the LeadsToCheckpoint data structure. For each method in LeadsToCheckpoint, all variable declarations are added to the variable table and any new data types are added to the mapping table. This algorithm will not be specified any further.

7.5.2 The Class Table

Only the user-defined classes that are being checkpointed needs to have the checkpoint and recover methods added. The mapping table holds all user-defined types that is checkpointed and hence this table can be used to fill the class table. The declaration of each user-defined type in the mapping table must be located and its variable names must be added to the class table. This is depicted in algorithm 8.

Algorithm 8 Algorithm for filling the class table.

FILL-CLASS-TABLE()

1: **for all** user-defined types T in $MappingTable$ **do**
2: locate declaration of T
3: **for all** variables X in T **do**
4: insert X into $ClassTable$
5: **end for**
6: **end for**

7.5.3 The Method Table

The Method table is also filled using the $LeadsToCheckpoint$ data structure. As with the filling of the variable and mapping tables, $LeadsToCheckpoint$ determines which methods that must be analyzed. Each method in $LeadsToCheckpoint$ is analyzed for invocations of any method in the $LeadsToCheckpoint$ data structure, as these method invocations must have inserted push and pop calls around it. Furthermore, the last variable declaration in these methods are found in order to determine where the recovery code is to be inserted. Algorithm 9 fills the method table.

Algorithm 9 Algorithm for filling the method table.

FILL-METHOD-TABLE()

1: **for all** entries A in $LeadsToCheckpoint$ **do**
2: locate declaration of A
3: $RecoverLocation \leftarrow$ location of last variable declaration in A
4: **for all** method invocations B in A **do**
5: **if** $B \in LeadsToCheckpoint$ **then**
6: insert $CurrentLocation, Method, RecoverLocation$ into $MethodTable$
7: **end if**
8: **end for**
9: **for all** checkpoints in A **do**
10: insert $CurrentLocation, Checkpoint, RecoverLocation$ into $MethodTable$
11: **end for**
12: **end for**

7.6 The Intermediate Format

The intermediate format must provide enough information to perform the analysis that must be carried out by the Source Code Analyzer. We have chosen to use an abstract syntax tree where all needed elements have their own node in the tree. The nodes must hold the information needed about the element it describes. A class diagram describing the intermediate format can be seen in figure 7.2.

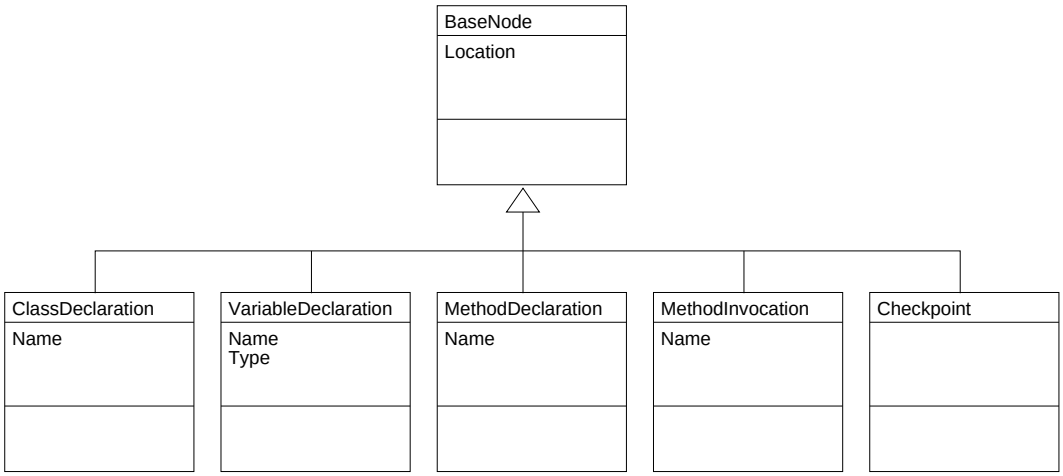


Figure 7.2: The class diagram for the intermediate format.

PART III

Implementation

The implementation part describes the parts of the design that have been implemented into the FROST system. Chapter 9 describes the implementation of the policies that control the migration procedure in FROST. In chapter 10 the tools for performing the checkpointing and migration of these checkpoints are described.

CHAPTER 8

Implementation Status

This chapter is a summary of the implemented system. We describe the differences between the design and implementation. The chapter will follow the overall layout from the design.

It should be noted that the current version of the FROST system is has only been tested on the Linux operating system and on Intel CPU architecture.

8.1 Policies

In the following we will consider the policies that have been implemented in the FROST system and discuss the differences that exist between the design and the implementation. Generally can be said that the overall functionality of the policies have been implemented while some of the more sophisticated features have been omitted due to time constraints.

Information Policy

In order to secure that the users always have priority over the FROST system, we have assigned priorities to the individual threads in the FROST system as described in the design. We have used the `nice` command for controlling the priorities of the FROST processes. The `nice` command enables us to lower the priorities of FROST processes in such a way that user processes can take over the processor whenever they are in need of it, thus ensuring that a user maintains full usage of his CPU. We have chosen only to assign priorities to the calculation code threads and the resource-checker thread because these threads are the ones that use the most processing power. The priorities assigned to the threads are chosen so that the FROST processes automatically yield to all the other processes.

As described in section 5.1.1, we use available resources as a measure for how loaded the machine is. We have chosen not to implement the use of memory in calculating the available resources on a machine based on the fact that the machines on which we are running the prototype of the FROST system has a large amount of memory and thus memory is not a problem at present. Instead we only use the average measure of the resources available to a thread with the same priority as a calculation code thread.

Transfer Policy

With regard to the transfer policy we have limited the implementation of it only to include a sender initiated approach and to using thresholds in determining whether a node is over-, under-, or average-loaded. The reason that only a limited version of the transfer policy described in section 5.2 has been implemented is due to time constraints.

Selection Policy

In section 5.3 of the design we describe a scoring system for selecting a process for migration. We also state that we choose only to use the time since the last checkpoint when selecting a process for migration. In order to enable the use of a more advanced scoring system, we have performed the implementation in such a way that it is easy to modify the implemented scoring system to include a more advanced system.

Location Policy

The location policy is dependent on the approach used by the transfer policy and thus we have also implemented a sender initiated approach in the location policy.

The protocols for the sender initiated location policy as described in section 5.4.2 have generally been implemented with the exception of the use of memory as a criterion. As described in the section above regarding the information policy, we have chosen not to use memory as a criterion in the selection process and thus there is no reason for providing information about the maximum memory limit allowed on a node prior to migration.

8.2 Migration

In chapter 6 the migration facilities of the FROST system is designed. In order to enable process migration all that has been designed has been implemented. This has been necessary as migrating a process is dependent on a number of things all touched upon in the design. We have, however, chosen to omit the support for recursive method calls. We see this as a reasonable limitation, because if a checkpoint is placed within a recursive call, the checkpoint that has to be produced will grow as it will have to include the data from all the previous layers as well.

8.3 Preprocessor

In section 7 the preprocessor used to ensure transparency in the FROST system is designed, but we have chosen not to implement it. This is primarily due to the time issue, and we do not find it is necessary to have an implementation of the preprocessor in order to show the functionality of process migration in FROST.

CHAPTER 9

Implementation of Policies

After discussing general ways of creating policies in section 3.4 and designing a set of policies specific to the FROST system in chapter 5, we set out to implement these policies. When implementing the policies into the already existing FROST system the view of the policies as four separate but cooperative entities disappear. As the policies are heavily dependent on each other it is natural that the edges between them seem to disappear in the implementation process. In the following sections we will try to uphold these edges in order to create a framework which is easy to compare with the framework set in chapter 5.

The policies are implemented in three classes, the Resource class, the CalculationCodeInfo class and the Master class, where the latter two are first introduced in [GK02]. The Resource class is a new addition to the FROST system, providing the main functionality of the information policy and acts as a toolbox for the transfer, selection, and location policies. In the following we will describe how the functionality of the policies has been implemented into the FROST system.

9.1 Information Policy

The information policy performs the evaluation of the available resources in the system and is described earlier in section 5.1. The main functionality of this policy is placed within the Resource class.

9.1.1 Threads and Timers

As described in section 3.4.1 the information policy is an information gathering process which collects information for the other policies so that they can make proper decisions based on the actual state of the system. The main task of the information policy is to gather information about the current state of the nodes in the system and store this information for future use.

In the design of the information policy we chose to include both the memory usage and the CPU usage in the calculation of the available resources in the system. We have however chosen not to include the memory usage in the implementation and therefore CPU usage is at present the only measure for available resources in the FROST system - the less available resources, the more loaded a node is. The memory usage should however be included in a final version of the system.

In order to enable the information policy to gather information in a realistic environment which reflects the number of tasks on a node, it is vital that the gathering commences on the same conditions given to the processes performing tasks on a node. We do this by letting the information policy run in its own thread at the same priority level as the tasks. This ensures that a higher priority process, such as a user process, can preempt the information policy thread, thus affecting the number of available resources and in this way represent a higher load in the system.

As discussed in section 5.1.1 we use a timer to control the runtime in which the number of available resources is measured. The timer is set and within its runtime the information policy thread performs the measuring of available resources. In section 5.1.2 we chose the interval with which the resource information is measured to one minute. We have chosen to set the timer to 100 ms and then obtain the resources a number of times for calculating an average.

The number of times the resources are measured could be changed in order to obtain an even more precise number for the available resources on the node. The problem is that the more times the thread performing the information gathering for the information policy is run, the more it would affect the actual purpose of the FROST system, namely performing calculations. We see that this

way of performing measurements of the available resources on a node are still vulnerable to random fluctuations in the number of available resources. This problem was earlier discussed in section 5.1.2 and we have implemented the solution to the problem given there, requiring that two intervals which follow each other has to show the same tendency in available resources in order to avoid unnecessary migrations.

In figure 9.1 can be seen an example on how using single measurements, as described above, can lead to unnecessary migration.

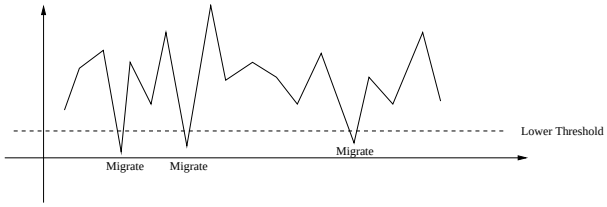


Figure 9.1: The effects of taking single measurements without smoothing.

As can be see from figure 9.1 taking single measurements is no good as the system reacts on load spikes and migrates processes more or less at random. Therefore the smoothing effect is advantageous. An example on this is given in figure 9.2

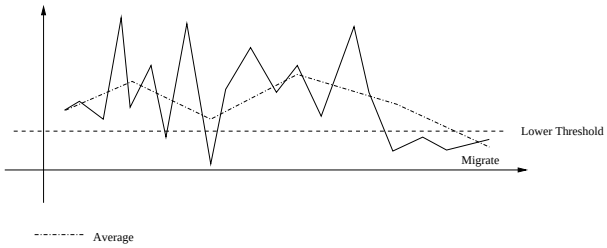


Figure 9.2: The effects of multiple measuring with smoothing.

In figure 9.2 no unnecessary migration takes place even though load spikes do occur. As the measurements in available resources decrease so does the average and a request to migrate process is issued.

9.1.2 Calculating Available Resources

The process of obtaining resource information on a node has previously been described in section 5.1.1 and the process will thus not be described any further. Instead we will concentrate on the problem of eliminating load spikes and how we have chosen to implement this in FROST.

This is done in the calculateResources method shown below. In this method we obtain the available resources a number of times and calculate the average value of the available resources in order to smoothen out the effect of potential load spikes. We call the number of times that the resources are checked for the resolution of the calculateResources method.

```
calculateResources ( resolution )
{
  for(int i = 0; i < resolution ; i++){
    set the timer;

    while(timer runs)
      nodeResources += Calculated resources;
```

```
    Sleep( runInterval / resolution );
  }

  return nodeResources /  resolution ;
}
```

This method is called with regular intervals and every time it has run we check whether the resource number has changed sufficiently to justify a broadcast to the other nodes in the system. We have chosen that the resource number must change at least 5% in order for it to be broadcast.

9.2 Transfer Policy

As described in section 5.2 the primary purpose of the transfer policy is to decide whether to take action or not based on the input from the information policy.

The transfer policy of the FROST system is very simple as it only decides whether the system state is overloaded or not. After the system state has been decided a decision about whether to act on the information is made. The system state is based on the average available resources for all the nodes in the system and is set by comparing the available resources of a node with a threshold which is also set by the transfer policy. When the state is set by the information policy, the master is notified and it takes further action if needed.

The transfer policy employs a sender initiated approach. Such an approach is described in section 5.2. The sender initiated approach states that there are two states that the system can be in. It can either be overloaded and thus try to migrate a process away or it can not be overloaded and wait for a process to be migrated to it. In the following we will describe the implementation of the transfer policy in greater detail.

9.2.1 Calculating Thresholds

After the information policy of a node has determined the number of available resources on that node the transfer policy calculates the average available resources in the system and sets a lower threshold representing the lower limit of the resources on this node. In the sender initiated approach only the lower threshold is used to determine whether it is necessary for the system to take any action. If the current resources on a node falls below the lower threshold, the system is overloaded and action needs to be taken to remedy that situation. In order to ensure that processes are only migrated if necessary, a node needs to be below the lower threshold, and thus be overloaded, two checks in a row. If this is the case the node will try to migrate a process away.

As described in section 5.2.1 the data used for the calculation of the average available resources from which the lower threshold is set, are not necessarily an image of the current number of available resources on a given node but only the latest resource information received from each node. We are aware of the problems introduced by using old data for performing these calculations, but we choose not to act on this as we see this problem as being small. This is due to the fact that the processes within the FROST system is in general long running and therefore the resource information should be relatively stable as long as a user does not interact with the system. As soon as a user chooses to interact with a node it could pose a problem not to announce the change in available resources of that node to the rest of the system because the other nodes could choose to act on old information and attempt to migrate a process to this node thus putting even more load on it. This could be avoided by either requesting the current number of available resources of a node prior to a process migration, as described in section 5.4.2, or by announcing the change in available resources of a node when a user starts to interact with it.

In the following we will describe the method which is used to set the threshold.

```
 calculateThreshold ()
{
  for( all  nodes  in  node_vector){
    tmpResource += presentNode->resources;
```

```
    }

    averageSystemResources = tmpResource/noNodes;

    systemThreshold.lowerThreshold =
                (averageSystemResources − LOWER_THRESHOLD_DELIMITER);
}
```

The average available resources in the system is calculated. Then we set the lower threshold relative to the average available resources in the system. The delimiter used for setting the thresholds should be a subject to future tests in order to optimize the system.

After the threshold have been set the transfer policy uses it to decide whether the system is currently overloaded or not by comparing the average available resources on the node with the threshold. As stated above, if the current resources of the node is below the lower threshold, the system is overloaded, and action should be taken to move a process away from the current node if it has any, and if the current resources are above the lower threshold the node is not overloaded and is thus a possible target for a migration from an overloaded node.

9.2.2 Acting upon Information

After the threshold have been set and the decision about whether the system is overloaded or not has been made, the master is notified. Here action is taken depending on the state of the system.

```
if (System State == OVERLOADED){
    ...
} else
    ...
}
```

The above code checks for the state of the system and acts upon this information. If the system discovers that it is overloaded it acts on this information but this is part of the selection and location policies and will therefore be discussed in the following sections. If the system is not overloaded, it takes no action at present.

9.3 Selection Policy

Where the transfer policy decides if a process should be moved or not the selection policy decides which process to move. The selection policy has been designed in section 5.3.

The tools of the selection policy is implemented in the CalculationCodeInfo class which contains information about all the different tasks currently running on a particular node.

The selection policy is enforced from the master and it is highly dependent on the state of the node. If the system is in a state where the amount of available resources is low and the transfer policy has decided that the system is in an overloaded state, the selection policy is executed. The selection policy chooses the process which is the last to have performed a checkpoint as the process that is to be migrated.

As the receiver initiated approach is not implemented into the system at present, no action is taken when a node is not in an overloaded state.

The selection policy of the FROST system consists of a single method which both produce the basis on which a selection can be made and performs the selection. In the following sections we will discuss this method in greater detail.

9.3.1 Selecting a Process

In order to make a selection of which process on a node that is best suited for migration we have decided to implement the approach mentioned in section 5.3.1. In section 5.3.1 we decided that the

time since the last checkpoint was performed by a process was to form the basis for the selection as we believe it to be the most important aspect in choosing a process. The reason for this choice is among other things that if the process which has last performed a checkpoint is chosen as little calculations as possible will be lost in the migration procedure.

As it is only a single parameter we are searching for we can look through all the processes and choose the process which has last performed a checkpoint. The code for this is shown below.

```
getProcess ()
{
  currentTime = time(0);
  tmpTime = -1;

  for ( all processes){

    if ( process->State != RUNNING){
      continue;
    }

    Time = currentTime - process->checkPointTime;
    if ( Time < tmpTime || tmpTime == -1){
      tmpTime = Time;
      processToMigrate = process;
    }
  }

return processToMigrate;
}
```

9.4 Location Policy

The location policy is the policy that decides which node a process is to be migrated to.

As stated in section 8.1 we have limited our implementation efforts only to include a sender initiated approach. This also goes for the location policy, and so apart from the fact that the receiver initiated approach has not been implemented, it is only the use of memory as a parameter to base the decision about whether to perform process migration that has been omitted from the location policy as it is described in section 5.4.

In the following we will describe the approach taken in implementing the location policy.

9.4.1 Choosing a Node

If the transfer policy in section 9.2 decides that a node is in an overloaded state the FROST system will try to migrate a process away in order to lessen the load on this node. In this case the location policy has to find a node which has the most available resources as it is likely to be the most capable of finishing a process the shortest time. We use a modified version of the prediction described in section 5.4.1 to find the node that will provide the best performance. The modification consists in that instead of sending information about number of processes and resources when a request is issued, these information is sent in the information policy along with the resources. The information is sent each time there is changes to the amount of resources, and therefore the information that is available at a node is maximally one resource-measuring interval old.

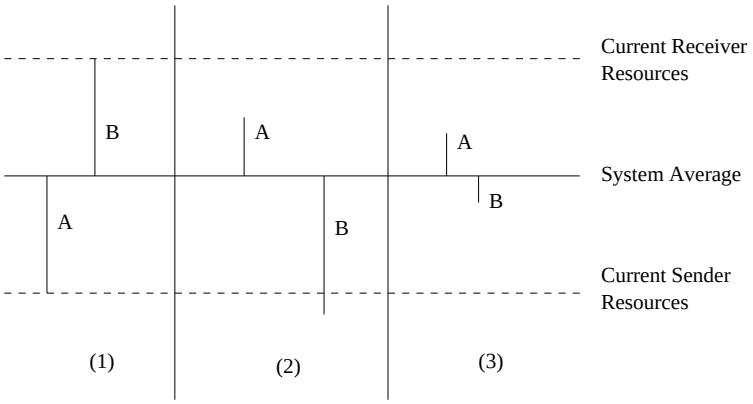
```
if (SystemState == OVERLOADED){
  ...
  if ( migrationInProgress != true){
    migrationInProgress = true;
```

```
    tmpRunning = localhost->runningProc;
    tmpResources = getResourcesOnLocalhost();
    tmpPredicted = ( tmpResources * tmpRunning)/(tmpRunning-1);

    For ( all known nodes){
      receiver_resources = node->resources;

      if ( receiver_resources > tmpResources){
        receiver_running = node->RunningProcesses;
        predicted_resources = ( receiver_resources * receiver_running )/( receiver_running +1);
      }

      if (( predicted_resources > ( tmpResources + 1500)) &&
          (tmpPredicted < receiver_resources )) {

        insertNodeIntoMap(node);
      }
    }
  }
}
```

First we record some information about the sending node including the predicted resources on the sending node after the migration. Then we look through all nodes that are known to the sending node. For each of these nodes we predict their resources after the migration. If the predicted resources for the receiving node is favorable we insert the node into an ordered map which orders the nodes after the resource numbers. In order to ensure that moving a process does not result in an overloaded node the sending node checks that the predicted resources of the receiving node is larger than those of the sending node plus a constant in order to ensure that better performance to the process is achieved. In addition the sending node checks whether the predicted resources for the sending node is smaller than the resources of the current resources of the receiving node. An example of this is shown in figure 9.3. Even though the total distance to the system average is less in figure 9.3(b), we do not allow migration as the predicted resources on node B will go below the current resources of node A.

Finally we get the node from the map that has the most predicted available resources after the migration and issues a request to that node.

If the request that is issued by the location policy is granted the process chosen by the selection policy is not sent immediately. Instead the system on the requested node verifies that it is not already executing a task from the chosen assignment. This is because the FROST system does not allow a process to migrate to a node which is already executing a task from that assignment, as described in section 5.4.2. If the requested node does not have a task from the same assignment, then the process chosen by the selection policy can be migrated to the new node. Otherwise the requesting node is denied its request and the next node in the map is chosen and a new request is issued.



A − Resources on the sending node
B − Resources on the receiving node

Figure 9.3: An illustration of the resources before and after a migration. (1) is the resource situation before a migration occur. (2) is an unwanted situation where the predicted values A and B show that A gains from migrating a process whereas B becomes overloaded and thus the system as a whole gains nothing. (3) shows a legal resource distribution where the overall resources in the system is closer to the system average after a migration.

CHAPTER 10

Checkpointing

In chapter 6 some classes and data structures were designed to handle the checkpointing and recovering of a process. In this chapter we will describe the implementation details of these classes and data structures and the extra features needed in the implementation process.

10.1 Control- and DataStack

The Control- and DataStack are very simple data structures and the implementation does not differ considerably from the design.

The ControlStack is implemented as a class with the methods described in section 6.1.1. For convenience, the ExecutionMode flag has been included in the ControlStack class as a public variable. There has not been made any further changes to the interface of the class. The constructor takes an instance of the DataStream class as a parameter which is used when checkpointing the class. By using the DataStream class, data marshaling is ensured and the control stack can be transferred across different architectures.

With regard to the data stack, we have chosen not to support recursive method calls. We believe that checkpointing in a recursive method call is often not reasonable. The deeper the recursion reaches, the more data needs to be saved, as all the data from previous calls is saved. Furthermore all recursive algorithms can be transformed to non-recursive algorithms [Sed92], and we therefore see it as an acceptable limitation. Hence, the DataStack is made as a simple array of integers as first described in section 6.2.2. All access to the data stack can be determined at compile time and should be inserted by the preprocessor.

10.2 CalculationCode Additions

In order to simplify the tasks that should be handled by the preprocessor, there has been made some additions to the CalculationCode class. Furthermore section 6.2.1 states that there must be taken special care of the CalculationCode, DataObject and ResultObject classes as they are already initialized upon recovery.

In order to handle these classes special Checkpoint and Recover methods have been made which saves and loads the member variables of the classes. To make sure that variables in the specialized calculation code classes are also saved, the methods are made virtual and any hence derived classes can implement these methods to save and load variables.

The Checkpoint and Recover methods are very simple in the CalculationCode class. There are only two simple variables, holding the total runtime that has been used to process the work unit and the number of active work units in the calculation code. Furthermore the methods must execute the Checkpoint and Recover methods in the DataObject and ResultObject classes respectively. This gives us the following Checkpoint method:

```
void CalculationCode::Checkpoint(){
  datastream « runTime « noOfActiveWorkUnits;
  mDataObj->Checkpoint(datastream);
  mResultObj->Checkpoint(datastream);
}
```

The DataObject and ResultObject classes does not have any knowledge of the data stream object used for the checkpointing, and it is therefore passed as a reference to the Checkpoint methods. The

Recover method is very similar to the Checkpoint method and will not be described any further.

The calls to the Checkpoint and Recover methods must be placed carefully. As will be explained further in the next section, the order in which data is saved and loaded must be exactly the same. Furthermore the member variables of the CalculationCode class must be instantiated before methods are executed in them. This is necessary as the checkpoint can be performed in any user-defined class, and hence such a class must be instantiated before the place of the checkpoint (and recovery) is reached. In the example code below, MyObject is a member variable of the specialized calculation code, and labels and goto's etc. have been left out. Furthermore the lines Checkpoint(); and Recover(); covers the entire checkpointing and recovering of data:

```
int MyCalculationCode:: calculate (){
   ...
  MyObject−>DoCalculations();
   ...
}

void MyObject::DoCalculations(){
   ...
  Checkpoint();
   If (ExecutionMode == RECOVER)
     Recover();
   ...
}
```

The call to DoCalculations cannot be executed before MyObject has been instantiated. It is therefore necessary that the member variables are recovered earlier than at the point of the call to Recover above. In order to ensure this, we have placed the call to the Recover method in the CalculationCode class even before the calculate method is started.

The same applies to variables declared at runtime, and recovery must therefore be done before a subsequent method call.

10.3 Checkpointing Data

The DataStream class was designed to handle saving and loading of checkpoint data using the XDR format. In order to handle this correctly, special data properties must be identified at runtime such as array sizes and pointer aliases. Furthermore, special care has be taken of user-defined classes.

To handle conversion of data into the XDR format we have chosen to use an existing library, providing us with these facilities. A short introduction to the XTL library and the facilities it provides are given in the next section.

10.3.1 The XTL Library

The XTL library[1] is a set of template classes that are designed to ease the task of converting C++ data structures into an independent format. Several formats are supported including the XDR format, and it also handles the saving and loading of data to and from disk.

Due to the way XTL handles the data that it stores it is vital that data structures are saved and restored in the same order. This means that if variable a is saved before variable b then variable a must also be restored before variable b. This is due to XTL using a file with sequential access and therefore everything is saved in consecutive order. When the data is restored, the file is read in the same order as when the data was saved and thus the data that was saved first is restored first.

The programmers interface has a number of different methods each handling a class of data types such as simple data types, arrays and STL-containers [Per99]. This has the effect that in order to save a variable, it is necessary to identify the type of the variable and recognize which class of variable

[1] `http://xtl.sourceforge.net/`

types it belongs to. This can be shown with the following example code, where stream is the XTL template class used for saving and loading data:

```
int i;
int a[10];
int *pi = new int;
int *pa = new int[10];
vector<int> v;
...
stream.simple(i).vector(a,10).pointer(pi).array(pa,10).container(v);
```

Furthermore it can be seen that the size of arrays must be declared when saving the variables. The library does handle pointer aliases but there are some limitations. For this reason and for the problems regarding variable types and array sizes we have chosen to implement the DataStream class as a wrapper class of the XTL library, taking care of these issues. The implementation details of the DataStream class is described in the following section.

10.3.2 The DataStream Class

The interface to the DataStream class was designed in section 6.2.1 to have Checkpoint and Recover methods for saving and loading data. To make it simple, these methods have been implemented by overloading the stream operators (« and »). This makes it possible to have the following syntax when checkpointing:

```
DataStream ds;

int a = 10;
int b = 20;

datastream « a « b;
```

When checkpointing values on the data stack, it is necessary to cast the addresses to the correct type, in order for the DataStream class to recognize them correctly:

```
datastream « *(int*)DataStack[0] « *(float*)DataStack[1];
```

The stream operators can be overloaded to handle a number of different data types, both simple types such as integers and floating point values and more advanced data structures. In order to reduce the complexity of the interface we have chosen to limit the set of data types that are supported. All simple types can easily be handled and are therefore supported. With regard to STL-containers we have chosen to limit the support to most of the containers that can be accessed with iterators. Furthermore, arrays and pointers are supported.

Most functionality of the class is taken care of by the original XTL library but some additions have been made both by adding functionality to the XTL library[2] but also by implementing the functionality in the DataStream class.

One of the problems mentioned in section 10.3.1 is that it is necessary to recognize which sort of variable type, e.g. simple, pointer or container, a variable belongs to, when inserting the checkpointing code. This is the main issue that has been solved with the DataStream class. The required functionality has been achieved by overloading the stream operators with a number of template functions each handling one or more types of variables. There has for example been implemented a template function for each container that is supported by the DataStream class. This limits the comprehensiveness of the class but provides an easy to use interface.

User-defined Classes

The XTL library also supports the handling of user-defined classes in the same way the DataStream class was designed to handle them. Hence, a method must be added to the user-defined class that

[2]The XTL library is open source software.

handles the saving of member variables in the class. We have chosen to use the exact syntax of the XTL library when adding these methods. In this way we do not need to handle user-defined objects in any special way in the DataStream class. When a user-defined object is to be saved with the XTL library it is done with the same methods as simple data types and pointers are saved with, and then the library automatically executes the saving function added to the user-defined class.

The functions that must be added to the user-defined classes needs to be template functions. This is due to the use of template classes in the XTL library, thereby making the format data is saved in, unknown until compile time. As we also wish to use the DataStream class for saving and loading data in user-defined classes and still let the XTL execute the methods automatically, it has been necessary to make the DataStream object globally available. If another solution should be used, it would be necessary to integrate the XTL library and the DataStream class far more, and thereby increasing the complexity considerably. As all checkpointing code is to be inserted by the preprocessor, we do not see this as a problem.

The following code shows an example on how the methods for saving and loading data is added to a class. In the XTL library these methods are called composite and takes the output or input stream as a parameter depending on the data is to be saved or loaded[3]. The stream is not used when we are using the DataStream class, but we need to use the exact XTL syntax as described above.

```
class MyClass{
  int val;

  template<class Format>
  void composite(obj_output<Format>& stream) {
    ds « val;
  }
  template<class Format>
  void composite(obj_input<Format>& stream) {
    ds » val;
  }
};
```

When an object is loaded with the XTL library it cannot provide the constructor with any parameters. For this reason it is necessary that each user-defined class which is to be checkpointed has a constructor that does not take any parameters. This also requires that references are not used as member variables if they cannot be set in the constructor without parameters.

Data Properties

As noted in the beginning of section 10.3 some data properties must be identified when checkpointing the data.

In the example in section 10.3.1 regarding the use of XTL, it was necessary to state the size of arrays when they were saved. In section 6.3 we stated that array sizes must be recognized in order to have all elements saved. By recognizing the array size automatically in the system, we can let the DataStream class use this information, so that we do not need to provide the size when saving data. This is done by overloading the **new**[] operator so that a data structure of allocated arrays and their sizes can be maintained.

Another property that must be recognized is pointers pointing to garbage. When the XTL library saves a pointer, it checks whether it is a null-pointer or not. This requires pointers to be set to 0 if they are not allocated. Declaration of pointers can be recognized and if they are not allocated at once they can be set to 0 instead. Furthermore deletion of pointers must be recognized in order to set the pointer to 0 after deletion. This is necessary as it otherwise would point to an address which is not allocated anymore. This is, however not adequate with regard to pointer aliases. It was previously stated that the XTL library takes care of pointer aliases for us, which is also true. But if the original pointer is deleted the alias will also point to garbage. The solution above where the deleted pointer is

[3] XTL uses one stream for saving and one for loading data. We have made the DataStream as a wrapper for both of these.

set to 0 does not apply in this situation, as pointer aliases cannot be recognized at compile time. The example code below shows the situation, where pb is a pointer alias to pa. It should be noted that it cannot be determined at compile time whether pa has been deleted or not.

```
int *pa = new int;
int *pb = pa;
 ...
delete pa;
pa = 0;
 ...
datastream << pa << pb;
```

Instead of setting pa to 0, we need a solution that also handles pointer aliases. By overloading the **delete** operator, we can keep track of all deleted data and thereby comparing pointers to the deleted addresses in order to locate pointers to garbage.

A more detailed description of the overloading of the memory allocation operators is described in the next section.

Due to the way the XTL library works there is an important issue with regard to pointers to stack variables. A pointer to a stack variable is not considered to be a pointer alias in the original implementation of the library. Due to this, the example below will not be handled correctly.

```
int a = 42;
int *pa = &a;
 ...
datastream << a << pa;
```

The value '42' will be saved two times, and when the checkpoint is reloaded, the second value will be allocated on the heap instead of creating the pointer alias correctly. We have corrected this flaw in the library by logging the address of all variables saved in order to identify pointer aliases. There is, however, still a limitation to the saving of variables. It is required that the stack variable is saved before the pointer to it. If not, the pointer will not be saved as a pointer alias to the stack variable, and thus it will be allocated on the heap upon loading, which is not desired. We have not implemented any solution to this problem.

Containers with Pointers

STL-containers holding pointers could not be saved using the XTL library. As this is a functionality that is useful in the FROST system, we have added it to the library, and provided template functions in the DataStream class for handling them. It has been implemented so that the pointers are included in the identifying of pointer aliases.

10.3.3 Memory Management

The overloading of the memory operators serves the purposes of identifying array sizes and pointer aliases to garbage. In the previous section it was stated that `new[]` was overloaded to log the array sizes and the **delete** operator to log deleted memory. It is obviously also necessary to log the deletion of arrays, as pointer aliases to arrays or elements inside arrays are very common.

Furthermore, there is a possibility that a previously deallocated memory segment is allocated again. If we only log deletion of pointers and not the allocation, a newly allocated pointer can be recognized as a pointer to garbage. By keeping track of all allocated memory, we can start by matching a pointer, that is to be saved, against the allocated memory. If the memory address is not allocated, we can search the deleted memory in order to see if the address has been deallocated. If it is not found there either, it is a pointer to a stack variable.

It is, however, possible that a previously deallocated memory segment is allocated again, but with another data type. If there exists a pointer to a non-existent user-defined object in the re-allocated memory segment, the XTL library will try to execute the composite method in the object that is

deleted. We have not solved this problem, but we have tried to minimize it by setting all garbage pointers to 0 when a checkpoint is performed. By doing this, we can erase the logged deallocations when a checkpoint is performed, thereby limiting the amount of data in the data structures used for logging.

PART IV

Test & Conclusion

This part contains test and conclusion to the introduction of process migration into the FROST system. In addition to that some thoughts regarding the scaling of FROST is given. Chapter 11 contains description and results of the tests made on the system, chapter 12 considers how the FROST system can be made to scale to a larger extent, and chapter 13 contains the conclusions to the project.

CHAPTER 11

Test

In this chapter we consider how the FROST system can be tested with regard to the process migration feature. We will first consider which tests to perform, then we will consider how these tests are to be performed in a manner that will give representative results. Last we discuss the results of the performed tests.

11.1 Test Types

When considering process migration in the FROST system there are a number of things which must be considered. In the following we will consider the elements which must be tested in order to determine whether it is an advantage to use process migration in FROST compared to not using process migration in FROST.

Correctness A necessary aspect when considering whether process migration has the potential to be an advantage in the FROST system is correctness. In section 3.5 we stated that the presence of process migration in FROST should not be detectable by the users. Therefore it is imperative that there are no difference in the result when using process migration as opposed to when process migration is not used. Furthermore FROST with process migration is not worth much if it generates flawed results. Note though, that testing for correctness only states whether process migration has a *potential* to be an advantage. Process migration may still show not to be an advantage based on some of the other test elements.

Performance The important issue when considering process migration in FROST is the additional time it will take for an assignment to complete after process migration has been implemented. It is obvious that if there is a loss of performance when using process migration, it may not be used by the users of the FROST system. It should be noted that the addition in time depends heavily on the assignment and therefore this time will differ depending on the period at which checkpoints are performed and how often a process is migrated etc.

Overhead In order to see how process migration affects the overall computation times of an assignment, it is necessary to find out how much overhead there is when performing a checkpoint. As the checkpointing code is performed on a regular basis the tests will show whether the checkpointing code has to be optimized or run with less regular intervals. We would like to find out whether the introduced overhead is too high to justify the use of fault-tolerance. The performance of the system is also degraded by the migration overhead. The migration overhead is the price for migrating a process and consists of the negotiation time between the sender and the receiver, the transferral of checkpoint data, and recovering from the checkpoint. This overhead will also be measured in order to determine the factors that influences the performance.

Transparency As stated in section 3.5 transparency from the users point of view is an important aspect. In order to maintain the ease of use of the FROST system API the user should not be required to consider the technical aspects of the process migration. As the transparency is thought to be provided by the preprocessor, which has not been implemented, we have chosen not to test this aspect.

Policies As described in chapter 9 we have implemented four policies in the FROST system. In order to ensure that the system performs as expected it is necessary to check whether the policies leads to thrashing. This is due to the fact that a system which is subjected to thrashing does not perform any calculations but merely moves processes back and forth between computers. As the purpose of the FROST system is to perform calculations it is imperative that this situation

does not occur, and that process migration does only occur when it is an advantage to move a process.

We have chosen not to perform any tests with regard to the scalability of the system and the effect of process migration when a larger amount of machines is introduced to the system. The system has not been designed to be a scalable solution as described in section 4.2.1 and it would therefore not be reasonable to test this issue. Furthermore these tests are used to measure process migration overhead and adjust the parameters used for the policies etc. We therefore need to be able to analyze the results from the different tests, which makes it necessary that we keep the results from the tests simple.

11.2 How to Perform the Tests

In this section we will describe the general issues that conform to all the tests that are described in the following sections. We will, however, start out by addressing the issue regarding correctness of the assignment results when using FROST with process migration. As noted earlier, this is a necessary aspect to fulfill as it forms the basis for the process migration feature in FROST.

11.2.1 Correctness

In order for process migration to be a viable feature in the FROST system, it is necessary that the assignment results are exactly the same and that they are correct whether the feature is used or not[1].

As process migration is performed exactly the same way independently of the tasks it is migrating, we believe that most errors can be detected by performing a limited number of practical tests. We have carried out a number of practical tests where we have compared the results of an assignment calculated both with and without the use of process migration. Different situations have been set up in order to test both one and several migrations between a number of machines. We have also tested that a task can migrate back and forth between two machines without changing the result of the assignment.

These tests should not be seen as a proof for the correctness of the process migration feature, but we find them adequate for the further testing of performance. We will not describe the testing of correctness any further.

11.2.2 User Processes

The purpose of FROST is to use the unused CPU-cycles of non-dedicated workstations. These workstations are normally used by a number of users who should not be interrupted or in other ways disturbed by the FROST system. A user may be typing in a text editor or compiling code. Neither should be disturbed by FROST. On the other hand, if a user uses a workstation intensively it might prove to be an advantage for the FROST system to move its processes away from that workstation in order to finish faster.

In order to test the overall performance of the system in realistic conditions we choose to create a user process which is supposed to simulate a user using a node. This process consists of a pattern of three types of intervals. Each interval i s10 minutes long and is either a series of load spikes which have a duration of 5 seconds followed by a 55 second sleep period, a long heavy usage of the processor which runs for 10 minutes, or a 10 minute sleep period where the node is completely free of load from the user process. The sleep period always comes in between the other two intervals and the only load that may occur in a sleep period comes from the processes in the FROST system.

Each node run one of these user processes and each of the processes have a unique pattern. This pattern makes it possible to predict with relative certainty when a process in FROST is going to migrate and where it is going to migrate to. In figure 11.1 the usage patterns are shown.

We have chosen to create these usage patterns in order to have a well defined usage of the processors

[1] In order for FROST to deliver similar results each time the same assignments is run it is necessary that the user does not use randomness in his calculations.

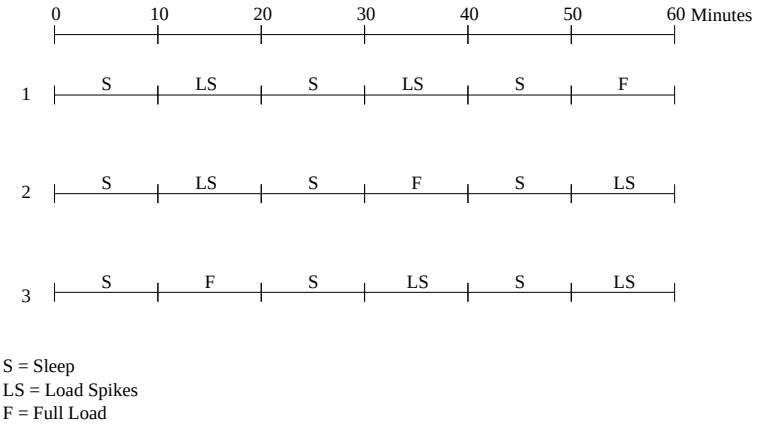


Figure 11.1: An illustration of the usage patterns in the user processes.

so that all the tests in this category can be run with the same resource characteristics. Using the same resource characteristics for each test enables us to compare the results of the tests.

During the tests we will run one user process on each node in the system in order to have changing load on the machines.

11.2.3 Test Problems

In order to obtain general results with the test cases mentioned above, it is necessary to test the system using multiple types of problems. These problems should range from small work units, small results and long term calculations to large work units, large results and relatively short calculations. The first type of test programs has a good calculation to communication ratio where the second type has a poor calculation to communication ratio. Both types of programs should be considered as the programs for which the system is used are not guaranteed to have an optimal relation between calculation and communication.

In section 1.1.2 in the introduction we have described a number of applications suitable to be solved in the FROST system, but as we do not have neither source code, algorithms nor data to calculate on available for any of these problems, we have chosen another problem type, raytracing.

Raytracing has proved to have the exact characteristics which makes a problem suitable to solve using the FROST system. It is a problem which is easily split into a number of tasks which are independent of each other. Raytracing is basically a number of light-rays being shot into a model from a camera and if the ray intersects with the model, a pixel is drawn. The color of the pixel depends on light sources, other objects, the surface of the objects[2] etc, and requires therefore extensive calculations.

Every pixel can essentially be computed independently of each other so that the problem is completely parallelized, all that needs to be distributed is the model with which the rays intersect and information about which pixels to calculate. Depending on the size of this model it may be advantageous to compute a number of pixels on each machine in order to keep the calculation to communication ratio at a reasonable level.

At present we have only implemented a single problem which supports process migration in the FROST system. This problem will be used to perform the initial tests of the system in order to determine if it performs reasonably. Further testing is required to obtain results which are representative of the system.

The purpose of our test problem is to simulate a realistic calculation assignment in the FROST system. Due to time limitations we will use a problem which is small compared to the assignments solved in the systems described in the introduction. In the raytracing problem we can adjust many parameters which makes it possible to optimize the problem in order to achieve the most precise results. These

[2] Reflectiveness, color etc.

parameters are described in the following:

Total runtime: The total runtime of an assignment will mostly be used when measuring the performance gain of an entire assignment. The total runtime of an assignment is measured from when the splitting of the assignment is started to all the partial results have been combined and the final result is ready.

Number of work units/Time per work unit: The number of work units seldom have a direct influence on the tests. It is mostly used to change the size of checkpoints as a smaller amount of work units will provide more data to save during calculations. The time per work unit is directly dependent on the number of work units.

Checkpoints per work unit/Time between checkpoints: The time between checkpoints is a very important measure, as it determines the time that can be lost at migration.

Checkpoint size: The size of checkpoints is also an important figure. It influences both the time it takes to perform a checkpoint and the time it takes to migrate a process.

Before each of the tests in the following sections we will describe the values used for these parameters where it influences the test.

11.2.4 Technical Specification

The system that the performance tests have been carried out on is a 7-node cluster. Each node in the cluster is a dual Pentium III 733 Mhz with 2Gb PC133 SDRAM ECC and they are connected through a fast switched Ethernet. All nodes are running Linux. Hence, it is a very homogeneous and dedicated system.

The FROST system is originally thought to run in a heterogeneous environment. We have although chosen to perform our tests in a homogeneous environment using dedicated machines and a dedicated network as it is easier to compare results that are obtained within such an environment. We do not, however, see it as a problem as all our tests are performed within the same environment and thus the results are comparable.

When running FROST on a dual processor machine with a single FROST task it has proven to be hard to make the task migrate while we are trying to simulate users in a realistic way. In order to remedy this situation we have created a process which takes over the unloaded processor during the tests. This has the effect that we are simulating a single processor machine.

We believe, however, that dual processor machines does not produce a problem for normal use of the system.

11.3 Overhead

We have chosen to start out by measuring the overhead that is induced by the different process migration features. First we will measure the overhead introduced by the checkpointing facility, and afterwards the migration overhead which denotes the price of a process migration. Finally we will measure the price for process migration feature in an ideal system, where there is no need for migrating any processes. The results from these test are to be used in the following tests regarding performance and stability of policies.

In general, there will not be running any user processes in the testing of overhead, as we wish to test how much overhead the process migration feature introduces itself.

11.3.1 Checkpoint Overhead

The checkpoint overhead is obtained by measuring the time it takes to perform a checkpoint in our test example. The time it takes to perform a checkpoint depends on two things. First, the amount of data that is to be saved is the main contributor to the overhead, as the data is saved in a file. Second,

the complexity of the data that is to be saved also influences the time it takes to save data. E.g. pointers must be analyzed in order to locate pointer aliases.

We believe that the checkpoint size is the most influential factor as it includes disk access, and we will therefore only vary this factor in the measuring of checkpoint overhead. When checkpointing in the test problem, we can vary this factor by changing the amount of pixels to be calculated in a work unit. This can be changed both by changing the horizontal resolution of the image and the number of rows in a work unit.

In this test we will not consider the entire runtime of an assignment, but only the time it takes to perform a checkpoint and compare it to the amount of data that is saved. This is useful as we cannot say anything in general about the runtime of an assignment compared to the amount of data that is contained in its checkpoints.

Table 11.1 shows the amount of data that is checkpointed in the test example. We have only calculated the amount of data that is directly imposed by the user, and not FROST control variables such as the extra variables in the work unit, that determines the type of data etc.

Data	Size (bytes)
Work unit	16
Result	depends on the work unit data
Temporary data	52

Table 11.1: The data that is checkpointed in our test example.

The size of the result can be calculated from the number of pixels that the work unit covers. The size of a result for a single work unit can be calculated using this formula:

$$ResultSize = h\_res \cdot no\_rows \cdot 3 \cdot 4,$$

where h_res is the horizontal resolution of the image and no_rows is the number of rows that is to be calculated in the current work unit. Hence, $h\_res \cdot no\_rows$ is the amount of pixels that is to be calculated. Each pixel consists of three colors[3] and each color is a four byte value[4]. The other parameters does not influence the result of this test directly and will be changed in order to achieve different checkpoint sizes.

From the results found in this test, we will setup expectations to the runtime of a complete assignment. From the size of the checkpoints generated by the assignment we can calculate the overhead introduced by the checkpointing feature.

By comparing this overhead with the difference between the total runtime and the runtime for the same assignment where all checkpointing and resource measurements are disabled, we can determine the price of the process migration feature if we are running in an ideal system where there is no need for migration.

Results

In this section we describe how we have varied the result size in order to measure the time it takes to perform a checkpoint. Table 11.2 holds the values regarding image resolution, number of rows per work unit, and the amount of data that is saved per checkpoint for each of the tests we have performed.

The result of the test can be seen in figure 11.2. It can be seen that the amount of time it takes to perform a checkpoint depends very linearly on the amount of data that is to be saved, and it takes about 1 second per 420 kb.

In order to set these results in perspective we will give an example. If we have a work unit that generates a 1Mb checkpoint the time for performing a checkpoint will be approximately 2.5 seconds.

[3]The RGB color scheme is used.

[4]The colors can actually be represented by two byte value, but the XTL library saves them as four byte integers anyway.

Resolution	Rows per work unit	Checkpoint size (kb)
160 x 120	12	22.6
800 x 600	10	93.8
1024 x 768	12	144.0
800 x 600	30	281.3
1024 x 768	32	384.1
3072 x 2304	12	432.1

Table 11.2: Specifications for the tests we have performed.

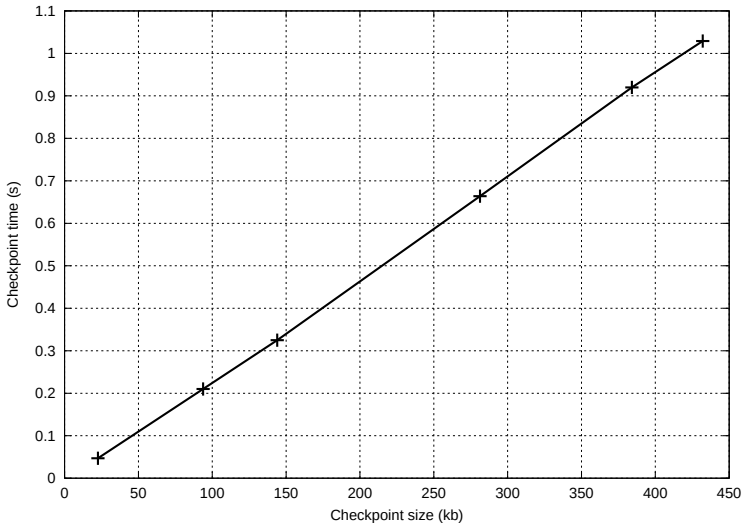


Figure 11.2: The result from measuring the time it takes to perform a checkpoint.

If it takes 10 hours to complete a work unit, and a checkpoint is performed every 4 minutes, the checkpointing feature will induce approximately 6 minutes overhead to the total calculation time of the work unit. We believe that a six minute overhead is quite acceptable as it gives the advantage that only six minutes of calculations can be lost, if a machine breaks down. A checkpoint for every 4 minutes is acceptable for fault-tolerance, but if the checkpoints are used for process migration the interval must be lower. From equation 4.1 in section 4.1.2 it can be calculated that 2.5 second checkpoint time requires a checkpoint every 95 seconds to justify a migration every 30 minutes. With a 95 second interval the overhead will be almost 16 minutes which is a 2.6% overhead to the total runtime. This is still an acceptable overhead but in order to gain an advantage of the increased overhead process migration must provide better performance than if migration is not used.

11.3.2 Migration Overhead

The migration overhead is the price that is paid when a process is migrated. In this test we will determine this overhead, but only the part that is induced due to actual migration, and not the part that consists in lost calculation time due to migration of an older checkpoint. That is, we wish to determine how much time it takes to migrate a process from one machine to another in the FROST system.

The overhead that this test will measure is very small compared to the total running time and therefore it is necessary to reduce any influencing elements. By reducing the total running time of the assignment we will reduce the probability for intervening load spikes. It will, however, also make such a load spike more influential on the running time, and we will therefore perform a number of runs for each measuring in order to determine an average value. Furthermore we will measure the time

per work unit instead of the total runtime and thereby removing influence from as much unimportant administration as possible.

We will conduct the test using only two nodes in the test system, and only one node is used for calculating the assignment at a time. First we will measure the work unit calculation times of the assignment using only a single node such that no migrations are performed. As the execution time of a work unit can vary, we will use the average value for all the work units. Afterwards we will make the same measurements but where a process is migrated back and forth between the two nodes a number of times. This test is also mostly influenced by the checkpoint size, and we have therefore varied this parameter.

As there are not running any user processes during this test, the processes will not migrate by themselves. We do not wish to add any load on the nodes as it will have influence on the runtime and thereby give erroneous results for the migration overhead. Therefore we have added constructs in the code that will force migration a fixed number of times, so that we can make comparable tests.

In the measuring of runtime without migration using a single node, all communication, such as transferral of work units and results, happens on the local node, and not over the network. When measuring the runtime when migration is performed, it is necessary to ensure that all communication, except for the migration, happens locally. Hence, the task must be on the same node as the master, when it returns a result. We will ensure this by always forcing an even number of migrations per work unit as shown in figure 11.3. Hence, the task will always be on the master, when a result is returned and a new work unit is received.

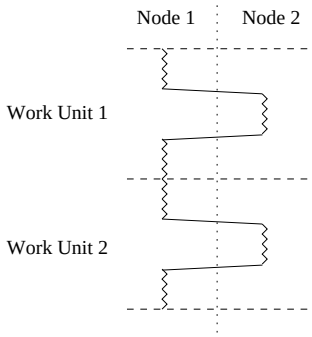


Figure 11.3: Migrations is performed an even amount of times in order to ensure that work units and results are only transferred locally.

Even though we are forcing migrations at predetermined times, we cannot ensure that no time is lost due to the time that has elapsed since last checkpoint. We will compensate for this by measuring the time since last checkpoint and subtract these values from the total runtime. The total price for migration will be measured in a later test.

Results

We have used the number of work units to vary the checkpoint size in this test. By varying the number of work units, we vary the size of the result that is calculated in each work unit and hence, the checkpoint size. The values used can be seen in table 11.3.

The result can be seen in figure 11.4, where the time per migration depends on the checkpoint size.

It can be seen that the migration price is linearly dependent on the checkpoint size except for the measuring with a 77 kb checkpoint size. We believe that this is a lower limit due to administration overhead. From the figure we can see that it takes approximately 1.2 second to migrate a 500 kb process and a 420 kb process can be migrated in 1 second.

Work units	Checkpoint size (kb)
48	77
24	154
12	308
6	614

Table 11.3: The values used for testing migration overhead. The checkpoint size depends on the number of work units.

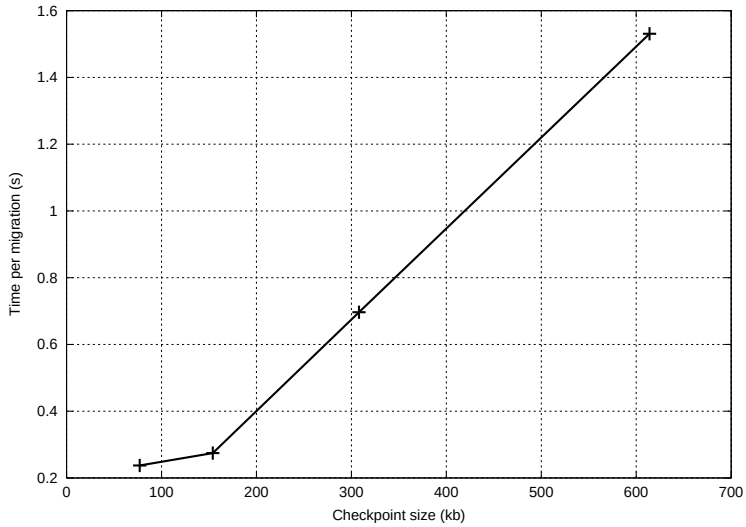


Figure 11.4:

11.3.3 General Overhead

From the two previous tests we can estimate the checkpoint and migration overhead for an assignment. If we compare the runtime for an assignment in FROST without any migration or checkpointing features with the runtime when the migration and checkpointing features enabled, we can determine any additional overhead due to these features. By comparing these two measurements without performing any migrations, we can determine the price of having process migration features in an ideal system, where there is no need for migration. We will measure this price in this test.

We have used the values in table 11.4 for the parameters described in section 11.2.3.

Parameter	Value
Work units (WU)	6
Average time per WU	32 min
Checkpoints per WU	40
Average time between checkpoints	49s
Checkpoint size	154kb

Table 11.4: The parameter values used for testing the general overhead induced by the process migration features.

We have aimed towards an average time between checkpoints of 38 seconds as specified in section 4.1.2. It is, however, difficult to adjust the parameters to an exact value due to differences in the work units. We have adjusted the average time to approximately 49 seconds and it ranges from 19 to 80

seconds.

From the results of the test regarding checkpoint overhead we can estimate that it takes approximately 0.36 second to perform a checkpoint of 154kb. With 40 checkpoints per work unit and 6 work units we have 240 checkpoints. The overhead induced by the checkpointing can therefore be estimated to:

$$240 \cdot 0.36s = 86.4s$$

Hence, we can expect more than 86 seconds overhead in the test with checkpoint and migration code compared to the test without.

Results

The runtime for the assignment without any checkpoint or migration features enabled was measured to 6043 seconds on average. This means that we at least can expect a runtime of 6130 seconds when checkpoint and migration features are enabled. The measuring of this runtime was, however, only 6122 seconds on average which is only 79 seconds overhead.

We believe that this is because the checkpoint and migration features induce such a small overhead that we cannot measure it precise enough with an assignment which has a runtime of 6000 seconds. Due to the low overhead of approximately 1.3% of the total runtime, we have not performed any tests in order to achieve a more precise result.

Hence, the result of this test shows that if process migration is not used, the overhead is not any larger than the overhead from the checkpointing feature, which still provides fault-tolerance.

11.4 Performance

The test of performance is very important to prove the applicability of the process migration features we have implemented in the FROST system. The performance test is very similar to the testing of general overhead, except that we allow migration in the second half of the test.

11.4.1 Total Runtime

The first part of the performance test consists in measuring the total runtime of an assignment with influence from user processes and with all checkpoint and migration features disabled. We simulate the user processes as described in section 11.2.2. Afterwards we perform the same test but with checkpoint and migration features enabled. The difference from the testing of general overhead lies first of all in the influence of user processes. Furthermore we add an extra node to the system which makes migration possible. We only use three nodes in total in order to simplify the output of the test so that we can analyze it in order to determine the stability of the policies.

Parameter	Value
Work units (WU)	12
Average time per WU	32 min
Checkpoints per WU	40
Average time between checkpoints	49s
Checkpoint size	154kb

Table 11.5: The parameter values used for testing the performance of the FROST system with process migration. The average times are measured with no load on the machine, and can therefore be higher during the testing.

The parameters used for the performance test is shown in table 11.5. We have increased the total runtime of the assignment to achieve a more realistic result. The runtime is, however, still only around 6.5 hours on a single node but due to time limitations we must limit the test in this way.

The first part of the test consist in measuring the total runtime without migration as described above. From the results of this test we will calculate an expected runtime of the assignment with the use of migration.

Results

The first part of the test gave a total runtime of 11608 seconds or 3 hours and 13 minutes using two nodes. When the assignment is run without any additional load, the runtime is approximately 9700 seconds. Hence, the load from the user processes induce around 1900 seconds overhead or almost 32 minutes. During a three hour period the simulated user processes will induce 60 minutes of heavy load across two nodes besides the load spikes. Hence, the calculation is not proceeding considerably during the heavy load.

When enabling process migration, a process should maximally run for two minutes during heavy load, before it is migrated. Hence, the heavy load will add $3 \cdot 2min = 6$ minutes to the total runtime. The 60 minutes consist in 30 minutes on each of the nodes. 30 minutes is $3 \cdot 10$ minutes, and hence, $3 \cdot 2$ minutes of runtime is lost. Furthermore, we expect a migration for each of the six times of heavy load where we loose $\frac{49}{2}$ seconds on average. The time added for performing the checkpoints consist of $12 \cdot 40 \cdot 0.36$ seconds, as we have 12 work units with 40 checkpoints each, and a single checkpoint takes approximately 0.36 seconds to perform. These overheads gives us the following expected runtime:

$$9700s + 3 \cdot 120s + 6 \cdot \frac{49}{2}s + 12 \cdot 40 \cdot 0.36s = 10379.8s$$

Hence, the migration feature should lower the total runtime of this assignment with approximately 20 minutes when user processes are introduced.

The result of the test turned out to be quite different though. The total runtime of the assignment with migration was 11858 seconds, 4 minutes more than without migration. The main reason for this added overhead can be found in the time that is lost due to migration of old checkpoints. The system migrated five times during the assignment and a total of 1080 seconds or 18 minutes was lost due to these migrations and a maximum of 350 seconds or almost 6 minutes was lost in a single migration. When there is no load on the nodes, there is only 49 seconds between checkpoints on average, but due to the low priority that the FROST assignments are running with, the user processes slows down the assignment and hence, the time between checkpoints increases. The measuring of resources should, however, be done every two minutes thereby only allowing an assignment to be slowed down for two minutes, but the thread measuring the resources has been given the same low priority as the assignments as described in section 9.1.1.

In the next section regarding the policies, we will analyze further on the results from this test in order to find ways of improving the performance.

11.5 Policies

In this section we will analyze the results from the performance test in order to determine if the parameters of the policies[5] are optimal or if the system is performing inappropriate migrations. First we will summarize the different parameters and the effect they have on the system behavior:

Interval: The interval between the measuring of resources has two effects. It decides the freshness of the data that is used for making migration decisions on the other nodes in the network. When a node has measured its own load, it will compare it to the last information received from the other nodes, and it will at most receive new information for each interval. It also decides how often migration will be considered on the local machine, and should therefore depend on how sensitive the system must be to load changes.

Runtime: The runtime of the resource measuring thread determines the width of the snapshot of the

[5] Resource measuring interval, resolution and runtime and the thresholds.

current resources. A longer runtime makes measuring less sensitive to small load spikes, but it will require more CPU-cycles for the measuring.

Resolution: The resolution is used to smoothen the load spikes as described in section 9.1.2. As the runtime is very short, a higher resolution can discover load spikes that are longer than the runtime.

Thresholds: The thresholds determines how willing a machine is to migrate a process. A large threshold will require the resources on a node to be farther away from the system average before migration is considered.

In the following we will determine if some of these parameters need to be optimized by analyzing the results from the performance test.

11.5.1 Analysis of the Performance Test Results

In order to analyze the results from the performance test we have plotted the measured resources, the migrations and the load from the user process into a single graph for each node. Figure 11.5 shows the resource graph for node 1 in the first performance test. The spikes and blocks at the bottom of the graph shows how the user processes act. The spikes indicates ten minutes of load spikes and the blocks indicate ten minutes of heavy load as described in section 11.2.2. In appendix B the resource graphs for all the performance tests can be found and we give a description of how the information used to make the graphs is obtained.

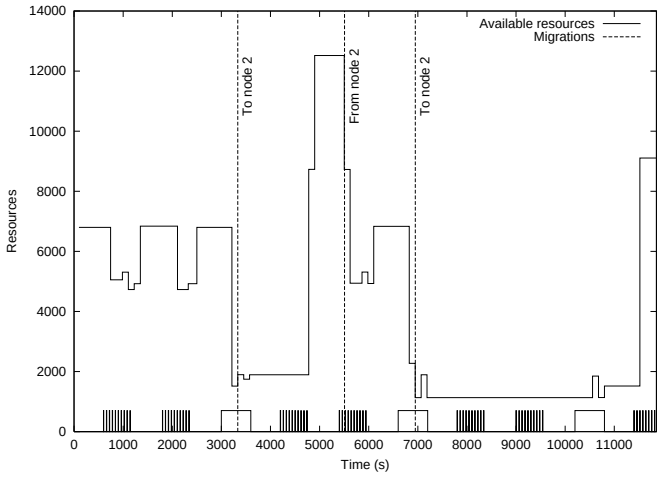


Figure 11.5: The graph for node 1 in the first performance test.

It can be seen from the figure that the policies react well on changes in the measured resources. When the first block of heavy load is encountered a process is migrated from node 1 to node 2. The resource graphs for all three nodes can be found in appendix B.1, where they can be easily compared. Figure B.2 shows that the process from node 1 is migrated to node 2 when there are lots or free resources on node 2. In general the migrations are performed with reason as processes are migrated from nodes with very low resources to nodes with plenty of resources. No unnecessary migrations are made taking the known resource information into account, and hence the implemented prediction scheme performs well.

The system reacts well to changes in the measured resources, but the figures in appendix B.1 also shows that the measured resources does not follow the added load from the user processes very well. There are sometimes rather large delays from the load from a user process changes until it is discovered by the system. In order to optimize this situation we will change the resource measuring interval to 30 seconds instead of one minute and carry out a new performance test.

11.5.2 Performance Test 2

The second performance test is carried out with the exact same parameters as described in section 11.4.1. The lower runtime interval for the resource measuring thread gives us a new expected runtime:

$$9700s + 3 \cdot 60s + 6 \cdot \frac{49}{2}s + 12 \cdot 40 \cdot 0.36s = 10199.8s$$

The total runtime in the second test was 11319 seconds which is an improvement of almost 9 minutes compared to the first test and almost 5 minutes better than the runtime when migration is not used. It is, however, still 18.7 minutes slower than the expected runtime. During the test there were five migrations which gave a total loss of more than 14 minutes with a maximum of more than 7 minutes for a single migration. The other four migrations were all above the expected average of 24.5 seconds.

The resource graphs for this test can be found in section B.2. They mostly shows a more correct measuring of the resources compared to the addition of load from the user processes. There is, however, still some delays in the measuring of resources.

Due to the large loss of calculation time when a process is migrated, it seems that the resource measuring thread does not run with the intervals specified. The entire execution of the resource measuring thread runs with lowest priority, which includes the part that determines the interval between the measurings. This can be the problem to our delayed measurings when the load is high on a node. We will perform a third performance test where we change the resource measuring thread to only run with a low priority when the actual measuring of resources is performed.

11.5.3 Performance Test 3

The third performance test was also carried out with the same parameters as the others. The parameters of the resource measuring test was also the same as in the second test, except for the priority. The changing of priority has not changed the measured resources but only increased the likelihood that the thread will run with the specified intervals. As no parameters are changed from the second test we can expect a runtime of approximately 10200 seconds.

The third test showed excellent performance with a runtime of 10370 seconds which is less than 3 minutes more than the expected runtime. Hence, the implemented process migration features shows very good performance as it decreases the total runtime with more than 20 minutes thereby removing 2/3 of the overhead induced by user processes when process migration is not used. There is however still possibilities for optimization.

If we look at the time that is lost due to migration of old checkpoints we see that 481 seconds is lost in 8 migrations. They all lies between 33 and 104 seconds which is more than the 24.5 seconds we have expected on average. If the loss per migration can be optimized to reach the average value, the total runtime can be decreased with almost 5 minutes giving us a total of 4% overhead compared to when process migration is not used in a non-dedicated system.

The resource graphs for this test is shown in appendix B.2 where they can be compared against each other. First of all they show that the measuring of resources is very realistic compared to the load induced by the user processes. Secondly, they show that the policies act very well, as processes are migrated away very shortly after the heavy load is started and never due to the load spikes. No unnecessary migrations are performed, except for the last from node 1 to node 2 which is just before the assignment ends, but we have not taken this into account in the policies.

11.6 Test Conclusion

In general the tests have shown good results. We have measured the overhead from the checkpointing feature and the price of migration which has proved to be fairly inexpensive compared to the possibilities that is provided. The performance tests showed a problem in the implementation of the resource measuring thread, but when it was corrected the system performed very well.

There is, however, one issue that should be considered. The overhead of checkpointing is small in the

sense of fault-tolerance, but if the amount of data to be checkpointed is large it induces a noticeable overhead. Furthermore the performance tests has proved that the time lost when a process is migrated is larger than expected mainly due to the decrease in execution speed when a node is heavily loaded. This increases the time between checkpoints and thereby the time that is lost when a process is migrated. As the process is migrated to a machine with more resources it will take the same time to catch up with the lost time, but the time used on the source node is still wasted.

In section 3.3 we stated that the use of migration points would not be an advantage in FROST as the time lost by migrating an old checkpoint was insignificant. The tests of the system has shown, however, that this is not entirely true. In section 11.3.1 we gave an example which showed that approximately ten minutes can be saved due to less frequent checkpointing when checkpoints are only used for fault-tolerance. Furthermore the time lost per migration would be $\frac{95}{2}$ seconds on average, which has shown to be higher in practice.

On the basis of these tests we believe that it will be an advantage to include the use of migration points to the FROST system. As noted earlier, migration points induce a much smaller overhead, when migration is not performed and therefore they can be placed much more frequently in the source code. When checkpoints are only used for fault-tolerance, they can be performed more seldom thereby inducing a smaller overhead. When using migration points there will not be lost any time when migrating due to calculations that need to be performed twice. The price of the migration will be the time to serialize and de-serialize data and transfer them to the destination node.

Tests have been performed which shows that the process migration feature can provide increase in performance when the system is used in a non-dedicated environment. Further testing needs to be performed in order to verify the results of these tests when the system is used with different types of assignments and with more random user processes. Furthermore it is necessary to test the performance when more nodes are introduced to the system. As described in the beginning of this chapter the system has not been designed to be a scalable solution and therefore testing the scalability at this point is not reasonable. The following chapter will consider the scalability issue in more depth.

CHAPTER 12

Scaling FROST

The philosophy of the FROST system is to provide a large amount of computer power without having access to a supercomputer. In order to be able to supply this amount of computing power a large number of computers must be used. The present version of the FROST system is only scalable to a local area network. A larger number of computers are most easily and cheaply accessed through the Internet where their unused computing cycles can be utilized in an approach resembling the SETI@home system. In SETI@home millions[1] of slaves connect to a single central master, obtain a work unit which they compute and then they return the results to the master. In order to make FROST an applicable system it is necessary to consider different ways of making the system scale Internet wide.

There are some very important differences between SETI@home-like systems and the FROST system which needs to be addressed when considering scaling in FROST. The main difference between FROST and SETI@home is that the FROST system allows all the slaves in the system to be masters in addition to slaves. This means that we cannot rely on a dedicated centralized server for administration of all the tasks in the system. Furthermore, there is no lower limit to the sizes of the computers in FROST. Any computer capable of connecting to the network and capable of running the FROST software is able to act as both a master and a slave. This also means that the FROST system cannot know anything about the computing power of a computer prior to it being introduced into the FROST system. It is a demand though that a master in the FROST system checks whether a node is capable of containing another task, with regard to memory usage and disk space, before it actually sends the task to that node. It is obvious that a task should not be sent to a node without room for another process. Additionally it is necessary to ensure that a master is always able to keep up with the data flow to and from itself and to store the data and the results of the assignments it is processing. If a master is not able to do this it should not be allowed to start any more assignments.

Another problem with scalability of the FROST system is that the current implementation uses broadcast to announce the availability of machines. Broadcast messages do not spread across the Internet and all machines will therefore not be reached. It would probably also pose a problem if they could be reached via broadcast as the amount of data needed to keep track of all machines would be enormous. It is, however, a problem if not even a single machine can be reached, or if the system is divided in subnets that cannot reach each other.

The solutions to make FROST scale can be identified as the following:

- Removing the bottleneck induced by the master
- Making a scalable information sharing solution

In the following we will consider how these solutions can be implemented in the FROST system, in order to make it scale to wide area networks.

12.1 Distributed Master

In order to remove the bottleneck that the master induces to the FROST system, we first need to identify the source of the problem. The problem lies in the masters ability to handle the communication if thousands of slaves are used in a calculation. All communication with the slaves goes through this single master which is a huge bottleneck either because of the network connection or the speed of the master, whichever is slowest. Hence, the problem can be identified as being the task of the master

[1] `http://setiathome.berkeley.edu/totals.html`

growing above the capacity of a single workstation. Of course this depends on the assignment being solved, but the more computers that is utilized for a single assignment the more the master will be a bottleneck.

The only solution to this problem is to distribute the master task to several machines[2]. In the FROST system all machines can already function as a master, but by distributing the master task is meant dividing a single assignment into a number of smaller parts and letting different masters take care of each of these parts. Hence, several machines act as a master for part of an assignment. This will share the communication between master and slaves between a number of machines, hence, reducing the bottleneck of the master. Eventually, the results must be returned to the master that started the assignment, but this can be done upon request of the master when it is ready for it.

Distributing the master task requires further analysis and design which we do not wish to address in this project, and therefore making the FROST system support a distributed master is laid out as further work.

12.2 Information Sharing

Better performance can be achieved by introducing more nodes in the system. The problem here is that the more nodes we introduce into the current implementation of the FROST system the more bandwidth we use for administrative tasks. The current implementation of FROST is limited to a local area network (LAN) on which we can use broadcast for sharing load information and for detecting new nodes in the system.

The possibilities for sharing information such as the state of machines[3] is very important to the scaling of FROST. As noted above, it is not realistic for each machine to hold information about all other machines in the system if it is to scale Internet-wide. Hence, the current usage of broadcast for sharing the availability of machines has two flaws. It fails in crossing network boundaries, thereby splitting the system, and it aims towards everyone holding information about everyone.

In the following sections we will consider new approaches to the way FROST communicates so that it may be made to run on a larger scale. Three approaches are given and discussed, a multi-cast approach, a multi-cast approach with partitions and a nearest-neighbor approach. Finally a short discussion of the problem imposed by firewalls is given.

12.2.1 Multicast

As stated above, the current implementation of FROST uses broadcast for communication. The problem of using broadcast for communication is that when scaling the FROST system to be usable Internet-wide a large part of the communication is going to be across subnet boundaries. As routers generally do not allow broadcast to pass a subnet boundary we are not able to keep in touch with all the nodes in the system, thus limiting the number of nodes to those that are reachable via broadcast. Therefore we have to consider an alternative way of distributing information within the FROST system.

The first solution that comes into mind is multicast as seen in figure 12.1. Multicast has the same advantages as broadcast, and if hardware that supports multicast is used, a multicast message does not bother computers which are not in the FROST system, otherwise it is necessary for each node to check whether a multicast message was meant for it or not. As most modern network interface cards supports a multicast filtering this is not a problem though[4].

Because of the great similarities between multicast and broadcast one could directly replace the broadcast methods with equal multicast methods.

Multicast provides better possibilities for a large number of nodes in the FROST system to talk directly to each other, but does, however, share some problems with broadcast. Multicast is supported in the Internet Protocol (IP), but as a best-effort protocol, where delivery to all members of a group

[2] Or add a server to the system handling the master assignment, but we have already excluded this solution.
[3] The state of a machine consists of whether it is online or offline and if it is online, the load of the machine.
[4] `http://www.erg.abdn.ac.uk/users/gorry/course/intro-pages/uni-b-mcast.html`

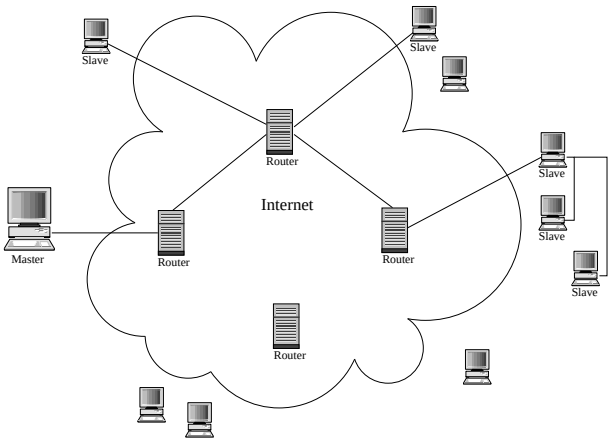


Figure 12.1: FROST using multicast across the Internet.

is not guaranteed [Tan96]. Furthermore, as the number of nodes increase in the FROST network the amount of load information and other types of communication handled by multicast increases. This has an effect similar to those encountered when using broadcast, namely the amounts of data to be handled gets too big. The data-structures containing information about all other nodes grows linearly with the number of nodes in the system. This may not sound as much but if millions of nodes are present in the network the amount of information contained on every computer about all the other computers is massive and table lookups would take longer and longer, thus rendering the system useless.

12.2.2 Multicast with Partitions

As stated above, multicast in its pure form has the same disadvantages as broadcast. But multicast is generally a good idea as every, or at least many, nodes in the system can be reached easily and the messages which are sent between different nodes does not need to be processed by computers which are not part of the FROST system. Because of these advantages it is worth reconsidering the multicast approach and finding a way to alter the communication patterns to introduce more flexibility into them. An approach which holds greater promise than pure multicast where everyone can talk to everyone, is to still use multicast but then split the system into partitions as shown in figure 12.2. Then a number of multicast channels could be used, one for communication between the routers and one internally in each of the different partitions.

A problem with this approach as well as the pure multicast approach is that in order to enable the nodes to communicate they have to agree on a multicast channel to use. There are several ways to solve this problem. One way is to obtain a permanent multicast address[5] and hardcode it into the system, but that solution presents more problems than advantages as it is not very flexible. Another way is to let the user specify the channels used. This approach is more flexible but it demands that a user knows the channels that are used by all the other nodes, and if the FROST system is to be used Internet-wide a multicast address has to be used that no one else uses. A third way of doing it is to allow the use of broadcasting inside a partition. The advantage here is that all nodes inside a partition can easily obtain knowledge about all the other nodes and thus agree on a multicast channel so that nodes which are not part of the FROST system are not bothered by the traffic. The communication between the routers present more of a problem as they cannot use broadcast as a means for agreeing on a multicast channel. A way to get around this could be to use a central master for exchanging the channel information. Such a master could be a common IRC-server[6] or it could be a dedicated master in the FROST system. The problem with a central master is that it limits the system by introducing a single point of failure so another approach than partitioned multicast might be worth considering.

[5] Internet Assigned Numbers Authority - `http://www.iana.org`
[6] `http://www.livinginternet.com/?r/rw.htm` and `http://rfc.sunsite.dk/rfc/rfc1459.html`

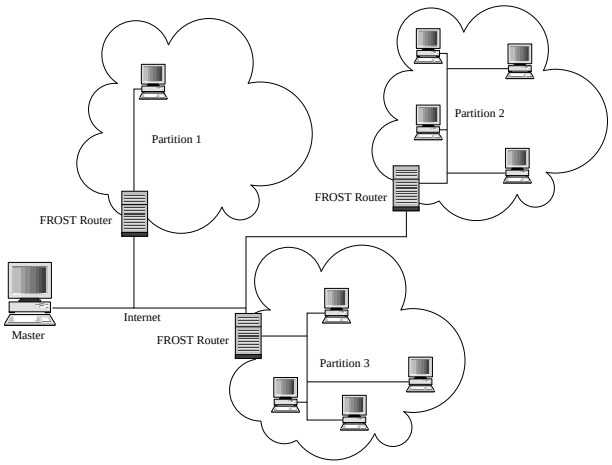


Figure 12.2: FROST using partitioning, routers and multicast.

An advantage with this solution is that the amount of data in the system is limited, as the FROST routers can collect the average load of the partitions and share the information with other partitions. The load of the individual machines will not be available across different partitions, but it is a reasonable limitation as we cannot expect to make decisions on information from thousands or millions of machines.

12.2.3 Nearest-Neighbor

If FROST is supposed to scale Internet-wide, we have to make some limitations. In the solution using multicast with partitions, the presence of routers in order to gather subnets that are not connected directly is required. Furthermore these routers have the task of gathering average load information from the different subnets, thereby limiting the amount of data in the system, but also removing some information with regard to the load on the individual machines. During a migration, this has the effect that we may not find the least loaded node in the entire system but rather a node that is sufficiently less loaded than the current node.

The nearest-neighbor approach is an approach which eliminates the problem introduced by using broadcast and multicast - flooding channels of communication. In addition to this it gets around the problem with a single point of failure introduced by using a single master in the partitioned multicast approach, but it will also make decisions based on a subset of the information about machines.

The main idea is that every node only knows about a relatively small number of other nodes which we call neighbors. Neighbors should only be located a few number of hops away from each other to reduce the amount of latency introduced if the distances are too great. A node can exchange information directly with its neighbors using unicast as shown in figure 12.3.

When a node gets overloaded it will send a request for a lesser loaded node, and add its own load to the request. Figure 12.4 shows an example, where a node with load 10 requests a lesser loaded node. It has chosen a hop count of 2. The hop count does not denote the number of hops through routers, but only hops through nodes in the FROST system. The FROST system could keep track of these hops by using a special hop field in each message. When a request arrives at a node, it decreases the hop count by one and retransmits the request, with the smallest of the local load and the load of the requesting node, if the maximum hop count is not reached. This can be seen in figure 12.4 (a). Replys are returned from nodes that has a smaller load than the requested. Each node only forwards the reply with the smallest load back to the requesting node as shown in figure 12.4 (b). The address of the node with the smallest load is also returned to the requesting node, so that the migrating process can be transferred directly to the destination node.

The nearest neighbor solution provides a scalable system where the scale of the system can be set

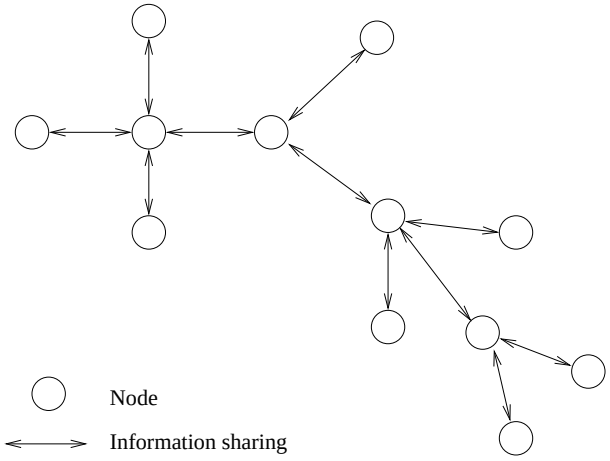


Figure 12.3: The nearest-neighbor approach.

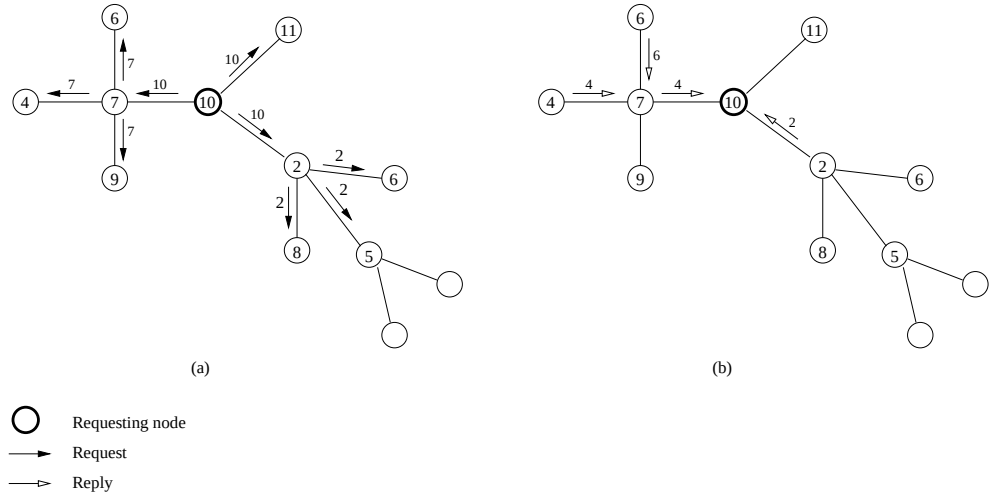


Figure 12.4: Finding a lesser loaded node in the nearest neighbor approach using a 2 node hop count. a) A request for a node with lesser load than the requesting node is rippled to nodes two hops away. b) Nodes only reply if their load is lesser than the requested.

dynamically by changing the hop count e.g. when requesting machines to solve the assignment. If a large assignment needs to be solved it can increase the hop count, thereby reaching a larger number of machines. Furthermore the amount of data transmitted between the machines is kept at a minimum, as only relevant data is transmitted. There are no need for using broadcast or multicast as all communication can be handled with unicast, but the solution requires that a small amount of neighbors is known.

There are some issues that need to be considered further, if this solution is to be used. E.g. if all neighboring nodes have a higher load than the requesting node, for how long should a node wait for an answer, how many hops should be used when sending out requests, and how do a node keep track of the online/offline state of the nodes it uses as slaves. As this is not the solution we consider in this project, these problems will not be addressed any further.

12.2.4 Firewalls

An additional problem when considering the scaling of FROST is that there is a firewall between most local networks and the Internet. We have to take into consideration how FROST nodes can be allowed to communicate through these firewalls. A possible solution to this is to use the SOAP protocol[7]. This is a protocol that allows FROST communication messages to be piggybacked on HTTP messages which can be sent through a firewall and authenticated using standard HTTP-authentication mechanisms. It requires that the firewall allows standard outgoing HTTP connections as described in Bolcer et al. [B+00]. They describe an approach where nodes behind firewalls use *Event Relays* as temporary storage of messages to computers behind other firewalls. Each computer then contacts this event relay regularly and requests any cached messages for it. The problem with event relays is that it requires a dedicated server to act as an event relay as the amount of communication with these relays can be large if a high number of computers wishes to connect to other computers through firewalls.

12.3 Summary

In the previous sections we have provided a number or possible solutions for making FROST a scalable system. In order to make FROST an applicable solution for the intended usage, some of these solutions must be implemented. Making a system scale Internet wide is a non-trivial issue which requires more research with regard to the FROST system. The above solutions is a step on the way with regard to moving the bottleneck imposed by the master and making the system cross network boundaries.

The distributed master and sharing of information is the two most important aspects of the scalability issues, as they form the basis of a scalable solution. By changing the way information about machines is shared, and the amount of information that is shared, e.g. with regard to number of known nodes per node, it is necessary to change some of the policies designed in this project. They are designed on the basis that all information about all the nodes in the system is known when decisions are taken.

Hence, making FROST a scalable system is very important to attain the intentions of the system but it requires some effort. Due to time limitations it has not been handled in this project and is therefore laid out as further work.

[7] `http://www.develop.com/soap/`

CHAPTER 13

Conclusion

In this project we have concentrated upon the design and implementation of process migration in FROST in order to obtain better performance when using non-dedicated machines and we have succeeded in incorporating process migration facilities into the existing version of the FROST system.

13.1 Design and Implementation

The process migration feature has been designed and implemented as a part of the FROST system. The migration facilities works in user space in order to ensure the possibility of a wider area of distribution as the process migration facility is a part of the application level program FROST. Additionally by placing it in user space we have made it possible to use an indirect extraction approach which is necessary in order to allow FROST to function in a heterogeneous environment.

The support for a heterogeneous environment requires special handling of the process state e.g. with regard to the runtime stack which can be very dependent on the machine architecture and the compiler used. For this purpose we have constructed the control stack where we keep information regarding method invocations and the point of execution from where the process must be restarted. The control stack can be transferred between machines independently of their architecture and operating system which makes it possible to continue execution from the exact point in the code where the checkpoint was made.

The indirect extraction approach requires insertion of checkpoint and recover code into the user source code and therefore it provides poor transparency. In order to remedy this situation a preprocessor has been designed which is thought to handle all insertion of checkpoint and recover code. In order to obtain an optimal placement of checkpoints in the user code we have chosen to let the user choose the points where they should be placed.

In order to provide possibility for fault-tolerance, which is important in a system performing long term calculations, we have implemented process migration using checkpoints.

A number of policies have been designed to secure good performance in relation to the expected use of the computers by the local users as we have described in section 4.1.2. The policies are the heart of the process migration features as they define how decisions are made when a process is to be migrated. A very important design aspect consist in ensuring that the system is stable and do not perform any unnecessary migrations. Still they must be sensitive enough to take action when a user adds heavy load to the node. Furthermore we needed to design a special way of measuring the load on a node due to the low priority that the FROST assignments are running with. In order to secure a stable system we have implemented the main parts of the designed policies.

In the following section we will describe the results of the design and implementation of process migration in FROST. We will both consider the results of the important decisions made in the design and the results from testing the system.

13.2 Results

The support for a heterogeneous environment in the process migration features has been fully implemented but it has not been tested across different architectures. There are, however, no architecture dependent elements and all data is saved in an architecture independent manner and so it should not be a problem to migrate processes between different architectures.

Tests have shown that the way load is measured in FROST, namely using available resources as described in section 5.1.1, is a good solution. This is both because it provides us with a valuable

tool on which further decisions can be based and because the way it is performed actually provides us with the resources that a process can expect to have access to. Furthermore the parameters of the resource measuring was tuned during the tests to give the system excellent performance as the system acts to change if it is needed assuming the user patterns we have described in section 11.2.2.

The prediction approach described in section 5.4.1 which is used to ensure that the FROST system is always in a better state after a migration is performed than it was before, has shown to be a valuable tool for the stability of the system. The most important issue is that it prevents unnecessary migrations when the idle load of two machines are equal. Without the prediction feature, a process would migrate back and forth between the two machines as the other machine would always be seen as having more available resources. The feature predicts that the resources will not be higher on the destination machine and therefore the migration is not performed. It should be noted that it is possible to create a user pattern which will make the system migrate back and forth e.g. by adding load spikes on the different nodes interchangeably just long enough to justify a migration. No matter which parameters we use in our policies, it is possible to construct such a user pattern. In order to remedy this situation adaptive policies should be considered. Such policies would react to the present user pattern instead of reacting according to a pre-determined plan and they would thus be much more flexible than our present solution.

We have carried out a few tests in order to check the correctness of the implemented features. The results of these tests have not been documented in this report but our general observation is that the features work as expected.

The results with regard to checkpoint and migration overhead shows a linear increase in time in relation to the size of the checkpoints which we believe is a reasonable result. The measuring of general overhead has shown that the price of the process migration features in an ideal system is no more than the overhead of checkpointing.

The performance of the system has been tested by simulating a non-dedicated environment. The first test offered poor performance as the total runtime of the assignment was four minutes slower than if process migration was not used. Through optimizations based on the results of the first tests we achieved a better performance with a runtime that was 20 minutes faster than the total runtime when process migration was not used. Due to the implemented prediction algorithm the system behaved very stable with no unnecessary migrations during the tests.

The conclusion to the tests was that it will be advantageous to implement migration points instead of using checkpoints for migration in order to achieve better performance, as it is possible to limit the overhead of the checkpointing feature and remove the overhead from performing the same calculations twice.

13.3 Further Work

In this project we have designed and implemented process migration in the FROST system. Our work has shown that in order to obtain an optimal system both from a users point of view and from a performance point of view there are some topics that need to be looked into. In the following we state these topics and why they must be considered. Furthermore we state some of the design issues that should be considered when designing the topics.

Migration points: The implementation of migration points into the FROST system should be considered. Both because our tests indicate that more performance can be gained but also because it is a fairly trivial task. When a migration point is used for process migration the procedure is very similar to performing a checkpoint. The only exception is that the migration point is transferred directly to another node where the checkpoint is saved to disk. The advantage of migration points over checkpoints is that the process state is only extracted when the system has chosen to perform a migration when using migration points, whereas checkpoints are performed fully every time they are encountered.

The use of migration does, however, also have a disadvantage as a node cannot migrate a process before a migration point is encountered even though it is heavily loaded. This means that if there is too long between migration points they have the opposite effect as the process

will receive less resources during a longer period and thereby decreasing the performance. As migration points introduce very little overhead into the system they can be inserted into the source code with more frequent intervals so that they are performed more often and thus raise the flexibility of the system.

Fault-tolerance: In order to fully implement fault-tolerance features into the FROST system it has to be ensured that the data-structures in the master and the slave containing information about the node and the tasks it is currently executing is saved. Additionally the fact that the hard disk on the slaves cannot be seen as stable storage for the checkpoints, must be considered. In order to ensure that these checkpoints are preserved in the best possible way a replication scheme, where checkpoints are distributed between masters, should be considered. In addition to this a protocol for handling machine breakdown is needed so that all nodes keeping checkpoints for the failing node can agree on who should restart the tasks.

Preprocessor: In this project we have designed a preprocessor for ensuring transparency of the process migration features in relation to user assignments. The preprocessor have been designed to support the insertion of checkpoints in recursive functions but at present it is not supported by the checkpointing facility which should be updated if recursive functions are used. If migration points are used it can be an advantage to support migration inside recursive methods. Implementing the preprocessor is a relatively simple task and it should therefore be done in order to make it easier for a user to create tasks for FROST. Including the locating of necessary variables should also be considered as an optimization to the system and it should thus be included into the design and implementation of the preprocessor.

Scaling: The present version of the system does not scale to an Internet wide platform due to the way it handles resource information and the way the master behaves with regard to general communication. The problem of making it scale is non-trivial and some considerations with regard to this is made in chapter 12. In order for the system to be widely usable it is necessary to make it scalable. When the system has been made scalable, additional tests should be performed with an increasing number of nodes in order to see how the system reacts to being scaled and in order to see if it affects the performance in any way.

Test of the system in a realistic environment: In order to ensure that the system is able to handle realistic user patterns without becoming unstable it has to be tested under realistic conditions. This can be done by installing FROST on several workstations in a normal office environment and test how the system reacts to the usage of the machines.

Bibliography

[ÁBL+95] José Nagib Cotrim Árabe, Adam Beguelin, Bruce Lowekamp, Erik Seligman, Mike Starkey, and Peter Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. Technical Report CS-95-137, 1995.

[App98] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, first edition, 1998.

[ARQ93] R. Andonov, F. Raimbault, and P. Quinton. Dynamic programming parallel implementation for the knapsack problem. Technical Report PI-740, IRISA, Campus de Beaulieu, Rennes, 1993.

[B+00] Gregory Alan Bolcer et al. Peer-to-Peer Architectures and the MAGI™ Open-Source Infrastructure. `http://www.endeavors.com/pdfs/ETI%20P2P%20white%20paper.pdf`, December 6th 2000.

[C+01] George Coulouris et al. *Distributed Systems - Concepts and Design*. Addison-Wesley, third edition, 2001.

[Cod93] P. D. Coddington. An analysis of distributed computing software and hardware for applications in computational physics. In *Proceedings of the Second International Symposium on High Performance Distributed Computing (HPDC-2)*, Spokane, WA, 1993.

[CS96] K. Chanchio and X. H. Sun. MpPVM: A Software System for Non-Dedicated Heterogeneous Computing. In *Proceedings of the International Conference on Parallel Processing*, August 1996.

[CTY99] W.F. Wong C.P. Tan and C.K. Yuen. tmPVM - task migratable PVM. In *Proceedings of the 2nd Merged Symposium IIPS/SPDP*, pages 196–202, 1999.

[ELZ86] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, 1986.

[FWM94] Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina. *Parallel Computing Works*. Morgan and Kaufmann, first edition, 1994.

[GK02] Michael Platz Glibstrup and Lars Kringelbach. FROST - A Distributed Heterogeneous Calculation Platform. Technical report, Aalborg University - Department of Computer Science, January 2002.

[HOS99] Janus Hardwick, Robert Oehmke, and Quentin F. Stout. A program for sequential allocation of three bernoulli populations. *Computational Statistics and Data Analysis*, 31:397–416, 1999.

[Kun91] Thomas Kunz. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. *IEEE Transactions on Software Engineering*, 17(7):725–730, July 1991.

[LTBL97] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report #1346, University of Wisconsin-Madison Computer Sciences, April 1997.

[MDP+00] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys (CSUR)*, 32(3):241–299, 2000.

[PE97] James S. Plank and Wael R. Elwasif. Experimental Assesment of Workstation Failures and Their Impact on Checkpointing Systems. Technical Report UT-CS-97-379, Department of Computer Science, University of Tennessee, 1997.

[Per99] Jost'e Orlando Pereira. XTL - The Externalization Template Library. Internet, `http://xtl.sourceforge.net/xtlguide.pdf`, 1999.

[SB94] E. Seligman and A. Beguelin. High-level fault tolerance in distributed programs. Technical Report CMU-CS-94-223, School of Computer, Science, Carnegie Mellon University, 1994.

[Sed92] Robert Sedgewick. *Algorithms in C++*. Addison Wesley, first edition, 1992.

[SH98] Peter Smith and Norman C. Hutchinson. Heterogeneous process migration: The Tui system. *Software Practice and Experience*, 28(6):611–639, 1998.

[SKS92] Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load Distributing for Locally Distributed Systems. *Computer*, 25(12):33–44, December 1992.

[Sri95] R. Srinivasan. RFC1832: XDR: External Data Representation Standard. Internet, `http://www.faqs.org/rfcs/rfc1832.html`, 1995.

[Sta98] William Stallings. *Operating Systems - Internals and Design Principles*. Prentice Hall, third edition, 1998.

[Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, third edition, 1996.

PART V

Appendix

APPENDIX A

Word List

This appendix explains some of the words used in the report. Some of the definitions are taken from [GK02].

Calculation code: The source code that is executed to carry out the assignment. It includes also the methods for splitting and combining the data.

Assignment: An assignment is the work that a user wants the FROST system to carry out. It is divided into several tasks that is distributed to a number of slaves. It contains the calculation code and the data that is to be calculated upon.

Task: A task is a part of an assignment that is to be solved on a single machine. During a task, a single work unit of data is processed.

Work unit: A work unit is the data that is used for processing a single task. It holds one or more data lumps.

Data lump: A data lump is the smallest amount of data distributed in the system. It contains one or more values of the same simple data type.

Simple data type: A simple data type is a non-composite type such as an integer or floating point.

APPENDIX B

Test Results

This appendix contains the resource graphs which show the results of the performance tests described in section 11.4.

The resource graphs shows the measured resources during an entire assignment. One resource graph for each node used in the tests has been made and hence, all three graphs from a test should be compared against each other. For convenience all graphs from a single test have been placed on the same page.

All Migrations to and from a node is indicated on the resource graphs so that it can be checked if there is any unnecessary migrations. In the bottom of each graph the pattern of the user processes is shown. Spikes denote load spikes and a block denotes that heavy load is added to the node. The user process patterns are thoroughly described in section 11.2.2.

The information needed for constructing the graphs has been collected by a logging process on a separate machine. Each node broadcast messages with the information needed such as measured resources and when checkpoints or migrations are performed which is then collected by the logging process. The resources are broadcast to the logging process at the same time as it is broadcast to the other nodes in the system, and the graphs therefore show the state of the system exactly as it is seen from a node in the system.

B.1 Performance Test 1

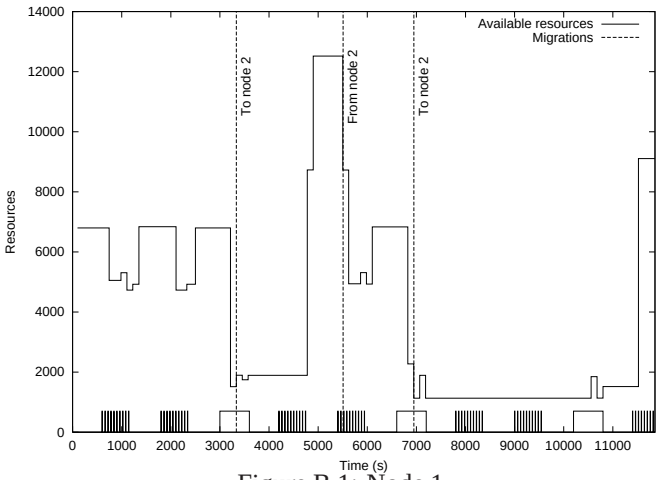


Figure B.1: Node 1

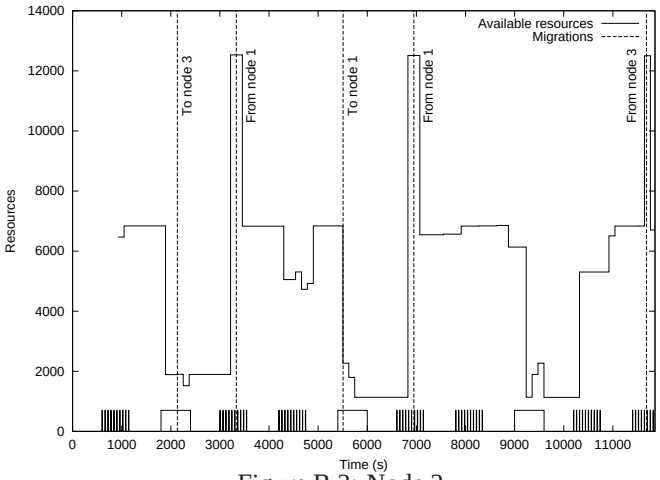


Figure B.2: Node 2

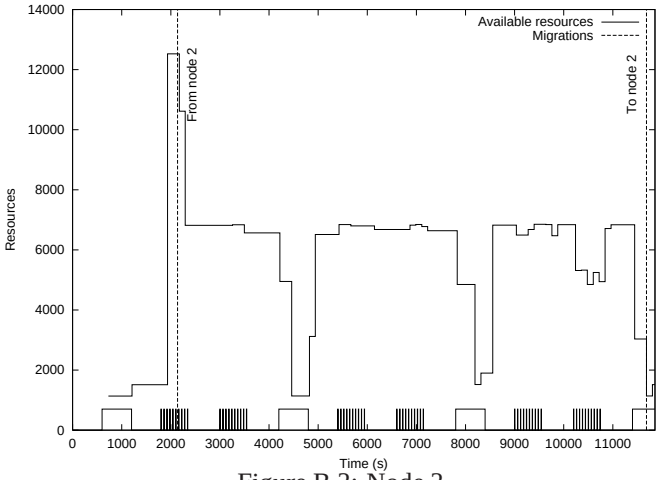


Figure B.3: Node 3

B.2 Performance Test 2

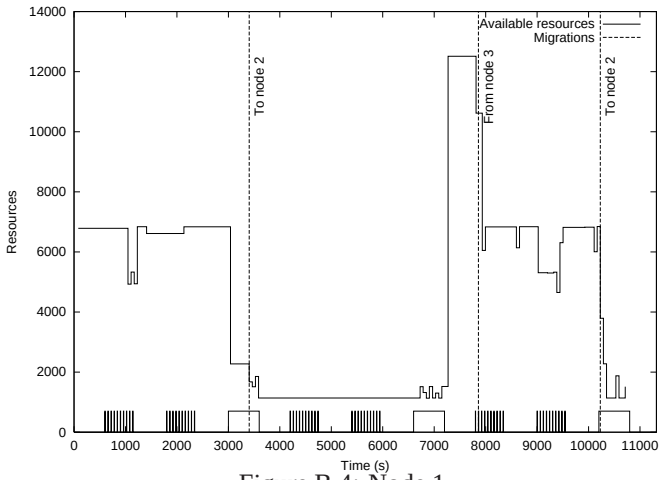


Figure B.4: Node 1

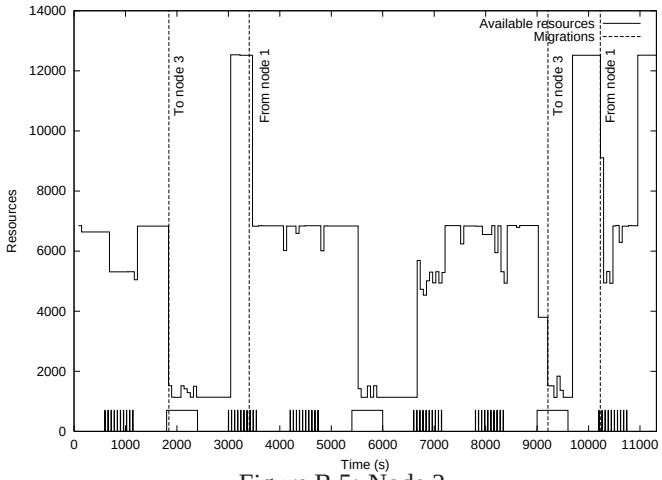


Figure B.5: Node 2

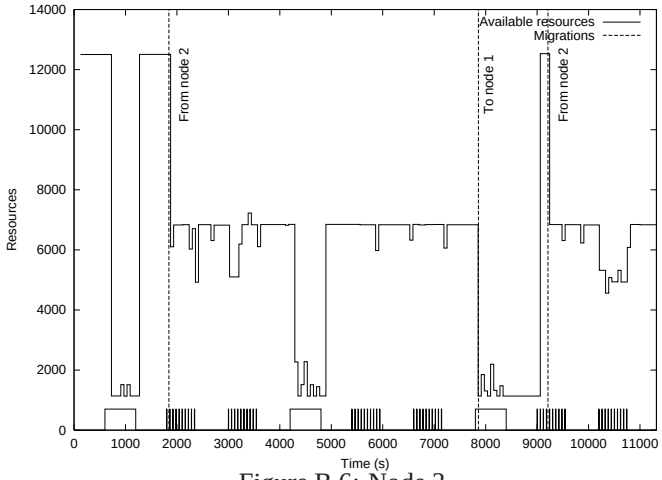


Figure B.6: Node 3

B.3 Performance Test 3

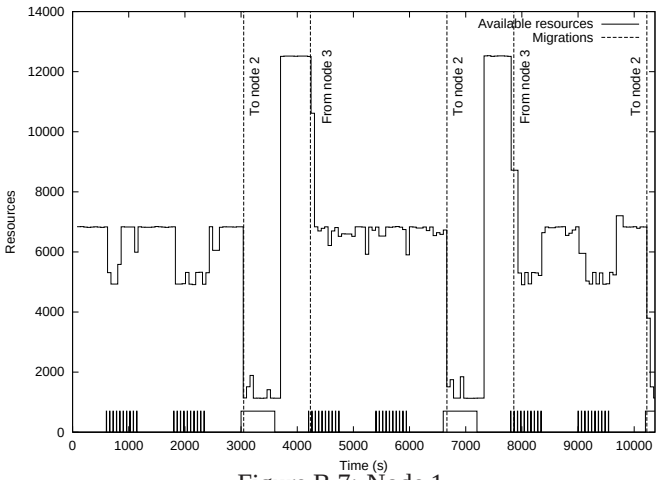


Figure B.7: Node 1

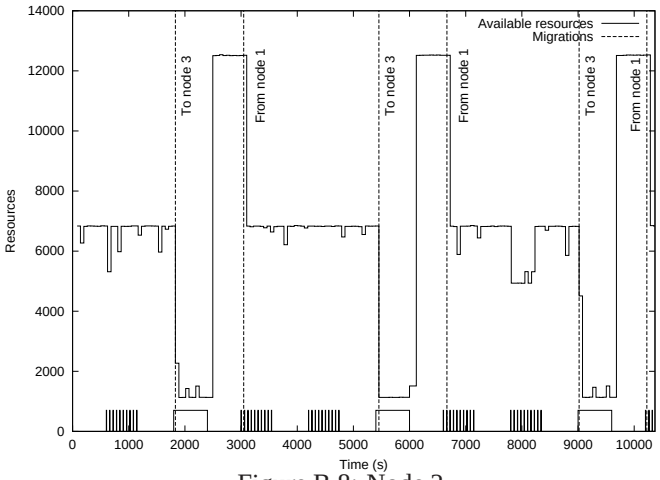


Figure B.8: Node 2

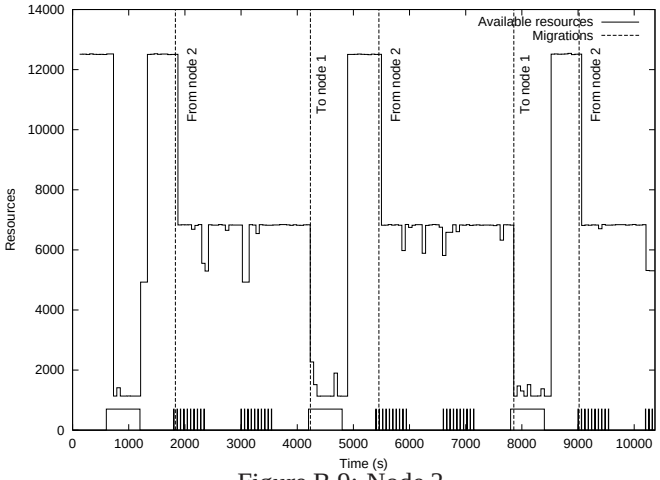


Figure B.9: Node 3