The Possibilities and Limitations of Heterogeneous Process Migration

by

Peter W. Smith

B.Sc (Hons), University of Canterbury, 1992

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

October 1997

In presenting this thesis/essay in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Computer Science
The University of British Columbia
2366 Main Mall
Vancouver, BC
Canada V6T 1Z4

Date: October 31st 1997

Abstract

Heterogeneous Process Migration is a technique that allows an active program to move between computers of differing architectures. While the program is executing, a migration tool will pause the program, locate the data values within the program's memory, convert them to a suitable format for the destination machine, then reconstruct the program on the destination machine so that it will continue executing correctly.

Although a small number of heterogeneous migration mechanisms have been proposed, few of them have been constructed, and none have yet resulted in a mature and efficient implementation. The Tui system has been constructed to provide an efficient migration tool for use on four common architectures within the Unix environment. Implementation lessons were learned while optimizing the Tui system to gain performance.

Tui has been used to derive a definition of *migratibility*. All other migration implementations have assumed that the program must be written in a type-safe language, or in a type-safe subset of a language. Since Tui has been designed to support heterogeneous migration of common languages that are non-type-safe, a survey of non-migratible language features has been undertaken. From this study, a definition for migratibility has been created, a framework for designing a migration tool has been given, and a comparison between migratibility and type-safety has shown that the two concepts are similar, yet different.

Contents

Acknowledgements

Throughout my four years at UBC, many people have come and gone in my life. It's hard to fully acknowledge everybody who has made a positive contribution towards this thesis.

The most thanks go to my supervisor, Norm Hutchinson, who always listened to my ideas, and carefully suggested how they could benefit from improvement. I appreciate how he let me work in my own way, rather than guiding me with a firm hand.

I would also like to thank my family for encouraging me to leave home and venture off to graduate school. Their diligent effort in learning how to use email, and writing to me every week, is very much appreciated.

Finally, the following people are also important, either because they have helped with technical advice, provided a pleasant working environment, or were just around when I needed them. In alphabetical order they are: Paul Ashton, Daniel Ayers, Ian Cavers, Tamarasi Dias, Margaret Dumont, Jean Forsythe, Joyce Furono, Mark Greenstreet, Scott Hazelhurst, Frank Henigman, Ed Knorr, Melany Lund, Holly Mitchell, Roxane Smyer, Alistair Veitch, Helene Wong and Ling Ling Yan.

PETER W. SMITH

The University of British Columbia

October 1997

To my Grandmother, Tui Hodgkinson.

Chapter 1

Introduction

1.1 What is Heterogeneous Process Migration?

Process Migration can be defined as the ability to move a currently executing process between different processors which are connected only by a network (that is, not using locally shared memory). The operating system of the originating machine must package the entire state of the process so that the destination machine may continue its execution. The process should not normally be concerned by any changes in its environment, other than in obtaining better performance.

Research into the field of process migration has concentrated on efficient exchange of the state information. For example, moving the memory pages of a process from the source machine to the destination, correctly capturing and restoring the state of the process (such as register contents), and ensuring that the communication links to and from the process are maintained. Careful design of an operating system's IPC mechanism can ease the migration of a process.

Most process migration systems make the assumption that the source and destination hosts have the same architecture. That is, their CPUs understand the same instruction set, and their operating systems have the same set of system calls and the same memory conventions. This allows state information to be copied

verbatim between the hosts, that is, no changes need to be made to the memory image.

Heterogeneous Process Migration removes this restriction, permitting the source and destination hosts to have differing architectures. In addition to the homogeneous migration issues, the mechanism must translate the process state so it may be understood by the destination machine. This translation requires knowledge of the type and location of all data values (in global variables, in stack frames, and on the heap).

1.2 Research Contributions

There are only a few heterogeneous migration systems in existence. Those that have actually been implemented are labelled as "prototypes" or "proof of concept". None have claimed to provide a complete and optimized migration mechanism. Further details of these systems will be given in chapter 2.

The major difficulty presented to any heterogeneous migration system is that all data values must be individually translated to a suitable format for the destination machine. This requires that the type of each data value, and its memory location be identified. Once the process has been moved from the source machine to the destination machine, each data value may need to be represented using a different storage format, and will almost certainly reside at a different memory location.

To locate data within a process, the migration system will use information generated by the source language compiler, and information maintained at run-time. Global variables are at well known locations within a program, and their type details are easily available. On the other hand, stack and heap values are more difficult to locate since they change dynamically during the execution of the program. Having knowledge of the underlying stack frame or heap space format is necessary to locate this data.

Previous migration systems have made the assumption that the source pro-

gram must be *type-safe*, or written in a type-safe language. This requirement simplifies translation since it guarantees that the program does not use any data values in a type-incorrect manner. If a variable was to be declared as one type, but the program is using it to store values of a different type, the migration system may incorrectly translate the value for the destination machine. For example, if a pointer value is stored within an integer-typed variable, the pointer value will not be correctly adjusted for use on the destination machine. Requiring that the source language is type-safe will ease the translation of data values.

There are two main contributions of this thesis. Firstly, a complete and optimized migration system was constructed. Building a complete system shows the true performance costs of heterogeneous migration, and uncovers new research issues that did not arise from previous, prototype, systems.

Secondly, the assumption of type-safety is not practical for existing software written in non-type-safe languages (such as C or FORTRAN), so a further investigation of *migratibility* has been made. This has resulted in a more precise understanding of the set of programs that can be successfully migrated.

This work has resulted in the following thesis statement:

> *Heterogeneous Process Migration is practical for programs written in common languages. The set of migratible programs is similar to but not equivalent to the set of type-safe programs.*

For this thesis, a heterogeneous migration system, called "Tui", was constructed. Tui is able to migrate programs written in the ANSI-C language between four different processor architectures: m68020 (a Sun 3), SPARC (a Sun 4), Intel x86 and Power PC, each executing a version of the Unix operating system. After analyzing the migration algorithms, Tui was optimized to reduce the time required to migrate a process.

Tui has been used as a platform for determining a precise definition of migratibility, rather than simply assuming that type-safety is a necessary property of a program that is to be migrated. The non-migratible features of ANSI-C have been analyzed, and have lead to a general discussion of the factors that influence whether or not a program can be migrated. These factors, and hence the set of migratible programs, have proven to be different to those of type-safety.

Finally, it is important to mention the issues that have not been addressed by this thesis. Any research problems that are normally associated with homogeneous process migration are assumed to have been solved by previous research. For example, the problems of moving kernel state and maintaining the communication links for a migrating process have been solved many times. Furthermore, we are only concerned with migration of a program's data, as the program's executable code does not normally change during execution. Program code can be made available for the destination machine by simply compiling the source code in advance.

1.3 Motivation

The traditional reasons for using process migration have been identified [38] as :

- Load Sharing among a pool of processors — For a process to obtain as much CPU time as possible, it must be executed on the processor that will provide the most instructions and I/O operations in the smallest amount of time. Often this will mean that the fastest processors as well as those executing a small number of jobs will be the most attractive. Migration allows a process to take advantage of underutilized resources in the system, by moving it to a suitable machine.

  It has been shown that load sharing is not always beneficial [25]. Since most processes only require a small amount of CPU time, with respect to the cost of migrating the process, there is no advantage to using migration over simply

executing a job locally or carefully choosing its initial machine. However, an important exception is for processes that require a large amount of processing time, for example, simulations.

- Improving communication performance — If a process requires frequent communication with other processes, the cost of this communication can be reduced by bringing the processes closer together. This is done by moving one of the communicating partners to the same CPU as the other (or perhaps to a nearby CPU).

- Availability — As machines in the network become unavailable, users would like their jobs to continue functioning correctly. Processes should be moved away from machines that are expected to be removed from service. In most situations, the loss of a process is simply an annoyance, but at other times it can be disastrous (such as in an air traffic control system).

- Reconfiguration — While administering a network of computers, it is often necessary to move services from one place to another (for example, a name server). It is undesirable to halt the system for a large amount of time in order to move a service. A transparent migration system will make changes of this kind unnoticeable.

- Utilizing special capabilities — If a process will benefit from the special capabilities of a particular machine, it should be executed on that machine. For example, a numerical program could benefit from the use of a special math coprocessor, or an array of processors in a supercomputer. Without some type of migration system, the user will be required to make their own decision on where to execute a process, without the ability to change the location during the lifetime of the process. Often users will not even be aware of their program's special needs.

Although process migration has successfully been implemented in several experimental operating systems, it has not become widely accepted. One reason is that mainstream platforms (such as MSDOS, Microsoft Windows and most variants of Unix), do not have sufficient operating system support for migration. Secondly, the benefits of using process migration are generally not great enough to justify the cost. That is, moving a process to another machine may be more costly than not moving it.

Recently, two new areas of computing have created new motivations for the use of process migration. Both these issues, *Mobile Computing* and *Wide Area Computing* will now be discussed in more detail. In both cases, heterogeneity plays a significant role.

1.3.1 Mobile Computing

Mobile Computing is a term used to describe the use of small personal computers that can easily be carried by a person, for example, a laptop or a hand-held computer. To make full use of these systems, the user needs to be able to communicate with larger machines without being physically connected to them - this is normally done via wireless LANs or cellular telephones.

It has been proposed [23] that process migration is important in this area. For example, a user may activate a program on their laptop, but in order to save battery power or to speed up processing, may later choose to transfer the running process onto a larger compute server. The process would be returned to the smaller machine to display results.

These concepts can be extended to allow a program to move between workstations as its owner moves. A person may be using a home computer, with a large number of windows on their screen. By remotely connecting to the computers at their place of work, they will be able to continue executing those programs in their office. If they choose to move between offices, the window system (and programs)

could follow them.

1.3.2 Wide Area Computing

For a computer to be part of the internet, it must understand the internet communication protocols. Since there are no constraints on other software, such as operating systems and programming languages, an enormous amount of heterogeneity exists.

The one limitation of global computing which will never be resolved is the propagation delay that is suffered over wide area networks. At best, data can only be transmitted at the speed of light, causing noticeable delays. If a program makes frequent use of remote data, its performance will suffer.

Process migration can help alleviate this problem by moving the program closer to the data, rather than moving the data to the program [35]. Typically, a program would start executing on the user's local machine. If it later makes frequent accesses to remote data, the migration system will reduce the delay by moving the process to a machine that is physically closer to the data. This makes the most sense in the case where the program is smaller than the data, or the data is accessed frequently.

Wide area processing is a topic that has already been addressed in the Java [29] and Telescript [43] languages. Java is most commonly used for transmission of programs using the World Wide Web. Although it supports remote method invocation, it does not currently support migration of active code. On the other hand, Telescript allows migration, as its primary purpose is for "agent" programs to move between sites. Because of migration, a Telescript program may complete its tasks while minimizing long distance communication costs.

1.4 Previously Identified Design Issues

Before discussing the details of this research, it is necessary to look at the various classes of Heterogeneous Migration or Mobility systems already in existence. The

discussion focusses on the unit of information being migrated and describes how that information can be moved. The design decisions made for Tui are presented in section 1.5.

Existing heterogeneous migration systems can each be classified into one of the following categories, based on the structure of the code and data being migrated.

1. **Passive object** – For example, in the Mermaid system [74]. The process (or object) being migrated contains only passive data. There is no executable code to be moved. This situation requires that data can be converted from the source machine's format to that of the destination machine.

2. **Active object, migrate when inactive** – For example, in the COMET system [47]. The process has executable code as well as data. Migration may only occur when the code is not active. For example, in an object based system, objects will remain inactive unless an outside agent requests some action. Assuming that migration only occurs during these idle periods, moving a process is simply a matter of translating data. It is assumed that the executable code is available on the destination machine.

3. **Active object, interpreted code** – For example, in the byte-code version of the Emerald system [38]. If a process is currently executing code by using an interpreter, moving the process involves translating the state of the interpreter and all the data values it may access. If these values (that is, variables, parameters, temporaries and other miscellaneous values on the call stack) are stored in a machine independent fashion, then migration is straight forward.

4. **Active object, native code** – For example, in the native-code Emerald system [62]. If the active program is compiled into native machine code, then fetching the active state is more difficult. Each machine has its own method of storing a program's values. Differences are obvious in the layout of each stack frame, the usage of registers and the structure of the executable code.

Other issues that may have to be addressed when designing heterogeneous migration systems include:

1. **Process Originated Migration** – The process being migrated makes the decision of when to migrate, rather than leaving the choice to an external agent such as the operating system. This ensures that the process is in a well known state (for example, during a call to a "migrate now" procedure), rather than in relatively random place within the code. Migrating while in a known state reduces the amount of information to be migrated, and simplifies its retrieval.

2. **Additional code within the migrating process** – The task of collecting data values from a process, and then restoring them, can often be simplified by adding code to the migratible program (explicitly by the programmer, or implicitly by the compiler). This can be in the form of a special library that must be linked with the program, or perhaps in the form of extra code at the beginning and end of each procedure. The problem of adding this code is the overhead of the additional execution time.

3. **Type-Safety** – All existing heterogeneous migration systems require that the migratible program be implemented in either a totally type-safe language or in a type-safe subset of a language. The migration algorithm requires complete knowledge of type information, usually generated by a compiler, to correctly marshall data for the destination machine. If the type data is inconsistent due to deficiencies in the implementation language, migration becomes more difficult or even impossible.

1.5 The Tui Approach

The approach taken by the Tui System focusses on supplying a migration mechanism suitable for general purpose use. By far the majority of existing software, and

programmer experience, is in traditional (and non-type-safe) languages such as C, COBOL and Fortran. Having the ability to migrate programs written in more common languages will make migration much more widely available.

For these languages, the data conversion component of the migration algorithm is relatively complex. It must allow for difficulties such as the misuse of pointers, type casting and lack of explicit type information. The less type-safe the language, the more difficult it becomes to locate, associate a type with, and then translate the data. These problems do not appear in type-safe languages.

Although it is possible to say that non-type-safe programming languages tend to generate non-migratible programs, it is useful to approach each problem on an individual basis. For example, a program written using C may be non-migratible due to the way that one small part of the program has been written. Rewriting this section of code in a different way will ensure that migration is possible. Alternatively, the language compiler and run-time system could generate extra type information to clarify the type of a piece of data.

The following example of C code demonstrates this:

```
main()
{
  union {
    int a;
    float b;
  } u;

  u.a = 1;
  u.b = 23.45;
}
```

Upon migrating this program, the migration system must be aware of the most recent assignment to the union. Either the integer 1 or the floating point number 23.45 is translated, but since they share the same memory location, the migration system does not know how to interpret the data. In this case, several solutions are possible:

- Modify the compiler to maintain a *union tag* that records which element of the union was most recently accessed.
- Internally convert unions into structs, so elements have distinct memory locations.
- In some cases, such as for the value 0, the internal representation may be the same for both data types, therefore there is no need to determine which type is correct.
- The programmer must refrain from using the union construct.

The aim of Tui has been to discover and attempt to solve the problems that make common languages unattractive for heterogeneous migration. The following goals have been followed as closely as possible during the construction and analysis of Tui.

1. To use the active object, native code approach.
2. To provide a general heterogeneous migration package capable of functioning on a wide range of common operating systems, CPU types, and programming languages. The major limitation being that only operating systems that already supply homogeneous migration will allow totally correct migration. The current implementation of Tui provides migration for the ANSI-C language within a UNIX environment, although future extensions would be possible.
3. To minimize the run time overhead of the process being migrated. It is preferable that the additional overhead due to migration is limited to compile time and migration time, rather than reducing the efficiency of the program during its normal execution period. However, it may be considered worthwhile to sacrifice a small amount of program efficiency if it allows a program to migrate that would otherwise be non-migratible.

4. The implementation language should not be restricted, unless totally necessary. In the previous example of the *union* declaration in ANSI C, converting the union definition to a similar struct definition is not permitted by the ANSI standard. However, performing this conversion will allow a much larger number of existing programs to be migratible.

5. The user should not be required to write extra code or directives to help the migration system. It is considered undesirable to ask the user to register the data types and values that need to be migrated. The determination of this information should be done automatically.

6. The system should not be totally process originated. As well as reducing the execution overhead, this also allows an external agent (for example, the operating system) to request that migration take place at any time. However, allowing the compiler to suggest suitable places to preempt the process will reduce the complexity of the migration algorithm.

7. To be as efficient as possible so that the migration cost is considered to be negligible.

8. To reserve the right to reject a process for migration, if it has been determined that the program is non-migratible. That is, when migration is initiated, but fails for some reason, the process should continue executing as if migration had not been attempted.

Although these particular goals have been followed in the construction of Tui, they are by no means an absolute requirement for heterogeneous process migration. When creating a migration system, the designer must make appropriate assumptions, based on their particular requirements. Although much of the work presented in this thesis is based around the assumptions made by Tui, chapter 5 discusses the extent of the design decisions that can be made.

1.6 Results

The remainder of this thesis is organized as follows:

- Chapter 2 discusses related work in the area of homogeneous and heterogeneous migration. Since heterogeneous migration is a fairly new topic, several other important topics, such as checkpointing and data marshalling, are also surveyed.

- Chapter 3 describes the implementation of Tui, with particular focus on the techniques that are specific to heterogeneous migration. Lessons were learned from the original implementation that lead to the creation of a revised version that has been optimized to reduce migration time overhead.

  It was determined that because the original version of Tui made no assumptions about the source program, it was not always possible to determine the type and size of dynamically created heap data. The revised version of the algorithm places restrictions on the programming language so that the necessary information can be gathered. Also, careful analysis of the algorithm and its data structures has resulted in a complexity (for traversing the memory image of a process) of between $O(n)$ and $O(nlogn)$, depending on the program being migrated.

- Chapter 4 lists the aspects of the ANSI-C language that do not permit Tui to successfully migrate a program. As well as by reading the official language description, the features were discovered by analyzing and migrating a set of example programs. These programs were not written for the purpose of migration, and are representative of the applications that would benefit from migration. The non-migratible language features of ANSI-C are classified into four different categories, based on the reason why they hinder migration.

- Chapter 5 uses the analysis of ANSI-C's non-migratible features to provide

a definition of migratibility, and to show how it differs from type-safety. Although these properties are very similar, each has its own specific definition and requirements, and there are exceptional programs that are in one set or the other, but not both.

There exists a subset of migratible programs that are not type-safe, since migratibility (unlike type-safety) is not an absolute property of the program's source code. We are able to add run-time code to the program, make modifications to the migration tool, or make assumptions about the differences between the source and destination machines, that will make a program migratible. With type-safety, run-time code can be added, but it can only warn the user when a safety violation occurs, rather than making the program type-safe.

One important observation is that every program will be migratible if we choose to provide an interpreter on the destination machine that is able to simulate the environment of the source machine. In this situation, the use of an extensive amount of run-time code will counteract any non-migratible features of the source language. Migratibility is dependent on the implementation of the program, not just the source code.

From the other point of view, there exist type-safe programs that are non-migratible because they do not take into account that the program must move between machines. For example, if the program relies on the size or format of the underlying architecture's data types, the program may be incorrect after migration.

- Chapter 6 discusses the issue of code optimization. In order for programs to execute efficiently, the source language compiler will attempt to optimize the code it generates, which may hinder the migration system's ability to translate a program. This chapter identifies why optimizations cause migration prob-

lems, demonstrates that problems caused by *machine independent* optimizations can be resolved easily, and then proposes some solutions for maintaining migratibility when *machine dependent* optimizations are used.

- Chapter 7 surveys the migratibility of other programming languages in comparison to the problems discovered in ANSI-C. Older languages, such as FORTRAN, contain non-migratible features that were not present in ANSI-C. Other languages (such as C++, Pascal, Ada, Scheme and Java) tend to have removed many non-migratible features or have added replacement features in order to obtain type-safety.

- Chapter 8 demonstrates that although heterogeneous process migration is not trivial in terms of performance cost, it is acceptable given the potential cost of not migrating a process. Firstly, the performance of the individual components of the Tui system were analyzed, which led to the optimization of the algorithm. Next, the complexity of the entire system has been determined by analyzing the migration time cost associated with several large sample programs.

  To show that migration is possible for realistic applications, a text editor and a matrix multiplication package have been migrated. Although the migration time cost is typically in the order of 10-30 seconds, these programs are intended to execute for long periods of time, so migration cost is negligible. Finally, migration is shown to be a beneficial option in comparison to the alternative of accessing files over an expensive network link.

- Chapter 9 provides conclusions about this research, and discusses future work.

- Appendix A provides Unix style manual pages for using the various programs that comprise the Tui system.

- Appendix B provides a description of how to prepare an ANSI-C program for

migration by Tui.

Chapter 2

Related Work

Until recently, heterogeneous process migration was considered an interesting topic, but no mature implementation had been developed. However, current interest in world-wide and mobile computing has led to several implementations, although they have only had limited use within research environments.

This chapter briefly lists previous work in homogeneous migration, both on a per-process and per-object basis. Next, a discussion of other heterogeneous migration systems will show the different range of approaches, in particular, how they compare to Tui. Finally, other supporting areas such as garbage collection and data marshalling will be surveyed.

2.1 Existing Migration Systems

Traditional Migration Systems

Process migration (in its homogeneous form) is not a new topic, and has been studied extensively since the late 1970s. Much of the previous research has involved finding new and improved methods of transferring the state of the process from one machine to another. Examples of homogeneous process migration systems are V [16][68], Charlotte [8], DEMOS/MP [51], Sprite [22], Condor [15] and Accent [73].

A good summary of these and other systems is given in [48] as well as a more recent survey in [46]

Object Mobility

The idea of process migration has been incorporated into distributed object oriented systems. However, it has become more relevant to migrate on a per-object basis (or in groups of objects), rather than moving a whole program. Migration in this form is more commonly known as *Mobility*, that is, the object is mobile. Examples of such systems are: Emerald [13] [38] [52], DOWL [6], DCE++ [54], COMET [47], and COOL [41].

Other Heterogeneous Migration Systems

All of the following systems support migration of native code across different architectures, however, they each differ from Tui in some significant way. Many of them solely support process-originated migration, whereas Tui also allows an external agent to make a migration request. Secondly, it is common to incorporate the data marshalling code into the migrating process itself (either created by the compiler, or specified by the programmer), whereas Tui is a completely separate program. Finally, Tui has addressed and resolved some of the type-safety issues that could limit migration in other systems.

Possibly the first heterogeneous migration system [56][24], developed at the University of Colorado at Colorado Springs, was a prototype built on top of the existing migration features of the V system. This system uses templates to describe the layout of the various memory segments. These (compiler created) templates specify the size of each data element (for example, whether it is a 2 byte or 4 byte integer) for global data, stack and heap blocks. The type templates are similar to, but simpler than the method required by Tui.

The primary limitation of this system is the assumption that data will reside

at exactly the same address on all architectures (possible in the V system). This simplifies migration since there is no requirement to adjust pointer values. However, data types must be of the same size, and data structures must be padded to the largest size required by any of the architectures.

Later work (related to this system) [10] has lead to a more formal analysis of the points at which a process may be migrated (known in Tui as *preemption points*). *Pointwise Equivalence* is required between two computations (that is, executable programs on different architectures) if migrating between the computations is to be guaranteed correct. They discuss issues relating to the granularity of migration points, especially in respect to optimization of a program's code. Placement of preemption points in Tui could benefit from this type of analysis.

A second system [67], that was never implemented, introduces the idea of migration by recompilation. At migration time, a source level program is automatically created, transferred to the destination machine, and then recompiled. When the program is executed, it restores the state of the process and then continues execution at the correct location.

The motivation for this method was that the machine dependent knowledge (such as register usage and stack frame layout) is already embedded into debuggers and compilers. Therefore, a process can be migrated to and from any architecture that supports commonly available debugging and compilation tools. The main disadvantage is that migration time is greatly increased because of the need for source code compilation.

Another approach [64] requires that the migratible program check a state variable at various points throughout its execution (specifically at the beginning and end of each procedure). If the state variable indicates that normal execution should occur, no migration code will be executed. However, when migration is requested, the variable is set to indicate that the contents of the currently active procedure should be saved to, or restored from, an alternate address space. A source

to source C language translator is used to insert the additional code.

The Emerald system [13] [38] [62] is an object oriented language and environment that permits fine-grained migration of native code objects. The Emerald compiler creates a template describing the internal structure of an object as well as the format of each method's stack frames. Whereas most heterogeneous migration systems make use of the C language, Emerald is itself a type-safe language, so the task of migration is more straightforward.

The HMF (Heterogeneous Migration Facility) system [12] requires the programmer to explicitly register the data to be migrated. The migration library is linked with the migrating program and provides procedures for registering data values (given their address and a type description), for initiating migration, and for converting to external data formats.

2.2 Related Topics

Since Heterogeneous Process Migration is a relatively new area of research, there are very few existing systems. However, it is important to understand a wide range of related techniques such as checkpointing, data marshalling and garbage collection. The concepts of distributed shared memory and binary translation are also relevant.

Checkpointing

Checkpointing and migration are very similar. The main difference is that checkpointing requires that a process can be restarted after a long period of time, whereas migration assumes that the current external state will not change. For example, a checkpointing system may need to rollback any files that were being written to. A migration system would assume that the files remained consistent.

In most cases, a checkpointing algorithm assumes that the process will be restarted on exactly the same machine that it started on. This implies that heterogeneity is not an issue. However, if we wish to restart it on a different machine,

with a different architecture, then the problem is identical to that of heterogeneous process migration.

Several checkpointing systems have been created for Unix systems [15] [49], but they only function in a homogeneous environment. The recent concept of "Memory Exclusion" [11] demonstrates that careful selection of data values to be saved can reduce the cost of checkpointing. Another system [50] divides programs into modules that can be individually checkpointed. Each module is initialized by supplying it with either a fresh (empty) checkpoint file, or a checkpoint file from a previous execution.

Data Marshalling Packages

For software that is expected to function correctly in a distributed environment, it is vital that the heterogeneity present in the data storage formats be taken into account. Any data that is externally visible must be in a form that all consumers can interpret.

Several general packages are available to automate the data translation process. Given some form of data description, these systems will generate suitable functions for translating between a machine's native data format and some intermediate format. Two of the most common systems are Sun's XDR [45] and ISO's ASN.1 [5].

Tui does not take advantage of any standard system, since the packaging of the whole data structure is handled as part of Tui's algorithm, and the translation of single data values is trivial in the four machines supported by Tui.

One solution [33] has addressed the issue of transmitting cyclic data structures within the CLU programming environment (XDR and ASN.1 cannot correctly deal with cycles). This problem is not relevant for Tui, as it has a special method for locating memory blocks.

Garbage Collection

A garbage collector is capable of scanning through the program's memory, searching for, and freeing areas that are no longer being used. A good overview of uniprocessor garbage collection methods is given in [71].

The initial prototype version of Tui made use of garbage collection techniques to help locate data. However, most existing garbage collection algorithms are not accurate enough to correctly migrate a program. In many cases, it is assumed that all data items are distinct (as in object oriented programming), and that marking the data is somehow possible. Also, it is necessary for pointers to be clearly identified in some manner (such as tagging), so they are not confused with other data values.

One system [14] allows garbage collection to function within C programs, but without proper type information, an educated guess must be made to identify pointers. Any pointer sized data value in a register or on the stack is considered to potentially be a pointer. The memory allocator is used to decide whether the value points to a valid memory block or not. The limitation of this system is that we can never be totally sure of whether a data item is a pointer, or simply an integer. Although an incorrect guess is not fatal for a garbage collection system, it will not suffice for a migrator.

In considering type-safety, [21] discusses garbage collection for Modula-3. They introduce the idea that although the source code for a program is type-safe, the compiled executable code may not be. For example, in an array that is not zero based, a "virtual array origin" pointer often refers to a memory location before the start of the array. This improves performance when calculating array offsets, but also gives a misleading indicator of which memory is in use.

A second important issue raised in this paper is that of determining how to locate the "derived" pointer variables at garbage collection time. That is, if one pointer variable is derived from a second pointer variable (for example, it may point to a field within an object), that derived pointer must be updated if the object is

relocated. This requirement is only an issue if objects are moved individually, as opposed to moving an entire program.

In a final paper [69], an efficient method of marshalling data structures via garbage collection techniques is discussed. This approach provides linear time collection (and hence transmission) of general graph structures. Unfortunately, their algorithm is not suitable for use in Tui, since it assumes that memory blocks can be reordered as well as corrupted (that is, marked with a forwarding address to indicate the new location).

Heterogeneous Distributed Shared Memory

The *Mermaid* system [72] [74] allows distributed shared memory (DSM) to function between heterogeneous machines. That is, a group of processes residing on different machines are able to share a consistent view of a segment of memory. Unlike traditional DSM, the machines may have different data formats, requiring that the segment is translated as it is moved between machines.

This system uses information provided by the compiler to determine the types of the data being shared. It then generates stubs to perform the necessary conversion. Using customized conversion code is said to be more efficient than using general conversion facilities such as XDR and ASN.1. Problems with unconvertible data values, pointer correctness and variations in data sizes are raised, but not addressed.

The methods used in Mermaid can be useful for process migration, although they are for a rather simplified environment. Primarily, Mermaid does not address the vital aspect of converting the active components of the process (such as registers and stack). Secondly, it is limited to a well defined segment of memory, rather than the whole process image.

Binary Translation

Binary Translation is a technique that is used to convert machine code from one architecture to another. For example, one of its main uses was in the introduction of DEC's Alpha processor [58]. There was a desire to convert existing VAX software to the Alpha platform, without using the original source code. Another system [57] emulates complex instruction set machines by using binary translation within a RISC environment. Finally, the Macintosh Application Environment [36] allows executable programs that were compiled for the older CISC-based Apple Macintosh computers to be emulated, or dynamically translated, for use on their newer RISC-based processors.

In the context of heterogeneous process migration, binary translation could be used to migrate the executable program code to a different architecture. Even though the simple solution of recompiling the program from the source code has been chosen, Tui could also take advantage of binary translation.

Chapter 3

Description of the Algorithm

Tui is able to migrate ANSI-C programs between four different architectures: Solaris executing on a SPARC processor (i.e. Sun 4), SunOS on an m68020 (i.e. Sun 3), Linux on an i486 and AIX on a PowerPC. The software has existed in both a prototype and revised form. It is important to note that several lessons were learned from the prototype, and led to the creation of the revised implementation. The prototype version made use of a garbage collection style algorithm for locating blocks of data to be migrated. Practical studies indicated that this method did not allow sufficient type information to be gathered and did not allow adequate performance, therefore motivating a second version.

This chapter gives a complete description of the revised Tui algorithm, with focus placed on the interesting features. First, an overview of the algorithm is given, with a details of how the four major components interact. Next, each of these components is described in greater detail. Even though this chapter only presents the second version in detail, a comparison between the two systems will be made in section 3.6.

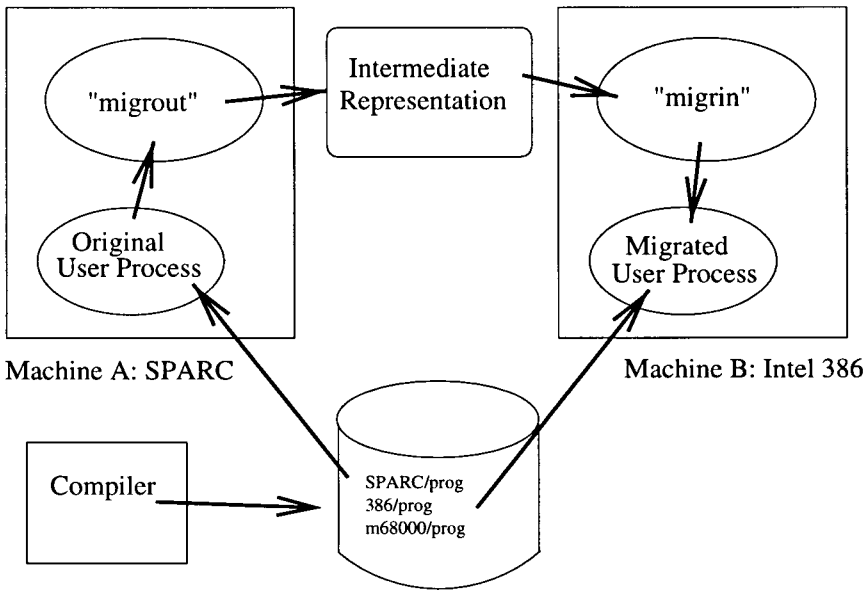

Figure 3.1: The Tui Migration System

3.1 Overview of Tui

Figure 3.1 shows how a process is migrated within the Tui environment. The following sequence of steps must occur for a program to be compiled, executed on the source machine, then migrated to a destination machine of a different architecture:

1. A program (written in ANSI-C) is compiled, once for each architecture. A modified version of the Amsterdam Compiler Kit (ACK) [66] is able to produce binaries for each of the four machine types supported by Tui.

2. The program is executed on the source machine, in the standard way (such as from the command line).

3. When the process has been selected for migration, the `migrout` program is called upon to checkpoint that process. Given the Process ID and the name of the executable file (containing type information), `migrout` will suspend the

process, fetch the memory image, and then scan the global variables, stack and heap to locate all data values. Next, all these values are converted into an intermediate form and sent to the destination machine. Finally, the process on the originating machine is destroyed such that the migration can no longer be aborted.

4. On the destination machine, the `migrin` program takes the intermediate representation and creates a new process. It is assumed that the program has been compiled for the target architecture so that the complete text segment, and type information for the data segment is available. After reconstructing the global variables, heap and stack, the process is restarted from the same point of execution as when it was checkpointed.

To make migration in Tui useful, an ANSI-C run time environment exists. Since each of the four architectures runs a different version of Unix, this library hides any inconsistencies. It was not possible to use each machine's standard set of libraries, as Tui requires that processes have the same view of the operating system on both the source and destination machines. The Tui ANSI-C library operates by directly accessing the machine's system calls. This library has not been modified in any way that would slow down the execution of a program, other than what was needed to make the code migratible.

Most variants of Unix do not allow migration, so movement of communication links and files (other than stdin and stdout) is not easy. However, a simple remote file server that allows migratible clients has been constructed.

The following sections describe the compiler and the executable files it produces, the `migrout` program, the `migrin` program, and the intermediate file format.

3.2 Compiler Requirements and Changes

To create programs that can be migrated by Tui, the compiler must ensure that sufficient type and location information is available to the other components of the system (`migrin` and `migrout`). Also, it must avoid generating code that is inherently non-migratible.

There were two main criteria for choosing a suitable compilation system. Firstly, the compiler must support a wide range of target architectures, and hopefully more than one source language. Secondly, the entire source code for the compiler, assembler and linker had to be available (for all architectures), so that modifications to their output could be made.

Three different compilers were considered. The `gcc` compiler [61] was the obvious choice as it can generate code for most common architectures. However, modifying the compiler and its related tools was considered too difficult due to the complexity of the source code. The `lcc` compiler [27] was considered, due to its wide range of target architectures and its ease of modification. However, it became obvious that important changes had to be made to the assembler and linker, which were not supplied as part of the package.

The compiler that was eventually chosen was ACK (Amsterdam Compiler Kit)[66]. This system is very easy to modify, and contains source code for all components. It has frontends for languages such as ANSI-C, Pascal, Modula-2 and Fortran, as well as backends for architectures such as SPARC, m68020, i386 and PowerPC. The major drawback of ACK is that it is only available at a cost.

If Tui were to become a viable commercial product, reverting back to `gcc`, and the associated `as` and `ld` programs, would be the best option. Not only is `gcc` one of the most widely available compilers, it supports a large range of architectures, and has many independent developers. It is quite likely that the modifications made to ACK could also be made to `gcc`, even though these changes would require a high degree of knowledge of `gcc`'s internal structures.

3.2.1 Features of ACK Generated Code

The structure of ACK has proven to be well suited to generating migratible code. It is desirable that an executable program have exactly the same structure on all target machines. That is, each compiled program contains the same set of symbols (procedure and variable names), and each procedure contains the same set of local variables and temporaries. The storage location and size of these entities may differ widely between machines, for example, local variables may be stored on the stack or in registers.

Since ACK frontends generate intermediate code [65], the differences between the various executable files is minimal. The majority of optimizations are performed on intermediate code, with the backends being primarily responsible for performing target instruction selection, as well as a small amount of peephole optimization. The optimization problems of code motion [62] are not relevant here.

ACK front ends generate *stabs* format [44] debugging information. These describe the type and location of all data values, using a compact ASCII encoding. Also, the mapping between source code line numbers and target machine addresses is recorded. Normally this information is used by debugging tools to allow the programmer to study an active program's data values. Tui uses these values in a similar, but more automatic fashion.

3.2.2 Modifications to ACK

The basic type information used by debuggers is not sufficient to correctly migrate a program. There are several important additions to the stabs format that Tui requires in order to successfully translate all data values. Aside from these additions, there are several other trivial modifications that were made (for example, the ACK backends were altered to correctly indicate which machine registers were used to store local variables).

The three major additions will now be discussed in more detail.

- **Preemption points.**

  When a process is migrated to a machine of a different architecture, we must deal with the fact that the corresponding point of execution (program counter) will have a different location within the text segment. To solve this, we select a set of logical points within the program at which migration is allowable. When performing the `migrout` operation to checkpoint a process, we must ensure execution stops at one of these *preemption points*. Upon restarting the process, the correct program counter value can be determined. Clearly, the program must have an identical set of preemption points on each target architecture.

  Placing preemption points within a program is an interesting issue. Points must be placed often enough so that the process will stop within an insignificant amount of time (excluding the possibility of system calls that could block). However, having too many preemption points will require an excessive amount of information, or may even lead to a situation where the process can not be started at an equivalent point. For example, if a preemption point for a SPARC processor is placed within a sequence of instructions that perform a multiply operation, there is no way of locating the corresponding point within the program on a PowerPC processor, since it only requires one instruction to perform multiplication.

  With these limitations in mind, it was decided that it is sufficient to place preemption points at the beginning of a loop, and at the end of each compound statement. The program will be halted within a very small amount time since no loop can repeat without passing through a preemption point (assuming the process was not blocked inside the operating system). Also, each machine's target optimizer is permitted to manipulate any code within a basic block, but

it must not move code across preemption points.

- **Call points**

  Although careful placement of preemption points can minimize the number of temporary values (partial results of a computation) that we must know about when the program is checkpointed, there is still the possibility that temporaries might exist across procedure calls. The following example illustrates this:

  ```
  x = foo(y) + bar(y)
  ```

  In this code fragment, the result of `foo(y)` needs to be saved somewhere while `bar(y)` is being calculated. However, if the process is preempted during the call to `bar`, it is necessary to retrieve the value of `foo(y)` from its temporary location (on the stack or in a register). Upon reconstructing the process at the target machine, the temporary is restored so that the calculation will complete correctly.

  This is achieved by generating a *call point* stabs at each procedure call. This specifies the address of the call instruction, the number of temporaries (partially evaluated expressions), the number of parameters being passed, and the type and location details of each of these values. Although the information about parameters is already specified as part of the callee's stabs information, there are some procedures (such as `printf`), where only the caller is aware of how many parameters are being passed and what their types are.

- **Stack frame details**

  During the `migrin` process, Tui must reconstruct each stack frame that existed before migration occurred. At compile time, a special stabs string is output at the beginning of each procedure. This specifies the size of the stack frame (that is, how many bytes are used for information such as local variables) as well as which registers were saved on the stack upon entry to that procedure.

3.3 "Migrout" : Checkpointing the Process

The `migrout` process is divided into four main phases. Firstly, the type and location information (generated by the compiler), is entered into Tui's internal data structures. Next, the migrating process is halted, and for easy access, a separate copy of the memory image is placed into Tui's address space. Thirdly, the type information is used as a guide for scanning this memory, and locating all data values. Next, these values are translated into an intermediate format for transmission to the `migrin` component of Tui. Finally, the process on the originating machine is destroyed.

3.3.1 Reading the Type Information

The stabs debugging information associated with a program is specified in a manner that follows the structure of that program. The executable file's symbol table contains a section for each *object file* (`.o` file) that makes up the executable. Within each section, the global variables and procedures are listed, with their appropriate type and location information. For procedures, the same type of information is given for parameters, locals and temporaries. Although the type information is specified in a one dimensional format within the file, Tui creates a multidimensional structure for internal use.

The stabs debugging format strings are converted (at compile time) into more appropriate type structures. These structures, known as *type graphs*, are similar to those used inside most compilers. They are able to represent all of the basic types as well as pointers, arrays and structures. To prevent name clashes, each symbol is prepended with the name of its enclosing file, and for local values, the procedure name.

Figure 3.2 shows the ASCII stabs strings for the given set of C declarations. It then shows the corresponding type graph entries.

In addition, two extra tables are required. The first table records the pre-

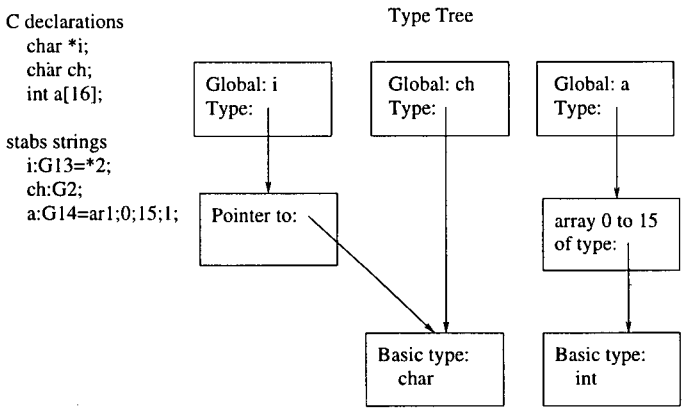

Figure 3.2: stabs strings and the type graph

emption points, each entry containing a single address for that point. The second table performs a similar operation, but for call points. In both cases, the table index is used as a machine independent representation of the point's address.

3.3.2 Halting the Process

Halting a program is more complex than in homogeneous migration, due to the difference between the source and destination instruction sets. The Unix `ptrace` system call is used to place the process into the `trace` state. `migrout` may now make a copy of the memory and registers. However, we must ensure that the process is in a consistent state (at a preemption point). The exact code for implementing this is machine dependent.

The current version of Tui stops the process, places a breakpoint instruction at *every* preemption point, then continues execution of the process until a breakpoint trap occurs. For large processes, it would be more efficient to insert only a few breakpoints (potentially only one), but it is not always easy to determine which preemption point will be reached next. For example, the use of asynchronous signal handling procedures, or function variables in general, complicate the prediction of

program flow.

As a final step, Tui fetches copies of the stack and data segments of the process, which includes the heap segment, into its own address space. The process can now be killed.

It is probable that altering the Unix kernel would allow Tui to have faster access to the information it needs, rather than using the `ptrace` system call. However, we have performed all of our research without modifying the operating system.

3.3.3 Scanning the Memory

While searching the memory of the process in order to locate all the data values, we must ensure that each value is detected exactly once. This is done by maintaining a *value table* that records the starting address, size and type of each piece of data. The value table is implemented as an expandable data structure where the only way to add a new value is to append it to the end. Therefore, the memory is scanned in a linear fashion, so that values are appended to the table in the correct order.

Firstly, the procedure entry points and global variables are scanned, and their details are entered into the value table. Global variables are very simple to handle since their locations are fixed and their types are well defined.

For the heap data to be scanned in linear order, it was necessary to alter the `malloc` and `free` memory allocation procedures so they would record all the blocks (empty and used) in linear order. This addition costs one extra pointer per memory block. Also, the compiler must generate a small amount of extra code for recording the data type of each block that is allocated. This issue will be discussed further in section 3.6.

Local variables (contained within stack frames) are scanned in a similar way. The frames are examined, starting at the most recent procedure activation. At each point, Tui queries the program's type information to obtain a list of the procedure's stack or register based values. Since stack based values are specified as offsets from

the procedure's frame pointer, their absolute addresses must be calculated. Special care is also taken to maintain a correct idea of the current register set, especially since registers are often saved on the stack across procedure calls.

At each point where a procedure call was made, Tui locates the associated call point information to determine which temporaries and arguments were stored on the stack for the duration of that call. A procedure's arguments will be scanned from the caller's perspective to correctly handle procedures that allow a variable number of arguments.

Finally, the command line arguments and environment variables are scanned. This must be done separately from the stack frames since this information is not always described by an explicit variable name as would a normal stack variable.

3.3.4 Marshalling to the Intermediate Form

The final stage of `migrout` is to traverse the value table and encode all data values from the memory of the process into the intermediate file. This potentially requires that data format conversion take place (for example, little endian to big endian integer formats). Section 3.5 gives full details of the intermediate file format.

The only difficulty in this phase is that we must represent the relationship between the different data items. That is, some data values will be (or will contain) pointers to other data values. When marshalling a pointer value, Tui performs a binary search on the value table to locate the information about the object being pointed to.

Each entry in the value table is assigned a unique number. When a reference is made to a data item, the pointer is encoded by specifying this machine independent number, rather than the machine specific address. Also, in the case where a pointer refers to a location that is part-way through a composite data item, an *offset* states how many indivisible subelements must be skipped in order to locate the correct value.

The following C code demonstrates:

```
{
  struct {
    int a;
    int b;
  } c[10];

  int *p = &c[2].b;
}
```

In this case, the offset for the pointer `p` would be 5, since the structure contains two subelements, and `p` refers to the second element of the third instance of that struct within the array `c`.

If Tui were to allow for the use of the union data type, the same system could be used, although since all elements start at a common memory location, it would only be necessary to count one data item per union.

3.4 "Migrin" : Reconstructing the Process

To restart a process on the destination machine, the `migrin` algorithm must obtain the program's type and location information in the same manner as for `migrout`. Next, it reads through the intermediate file and places all the data values in their appropriate locations. This phase reads the intermediate file sequentially, and can therefore be mostly done in parallel with the `migrout` phase.

Global variables are placed directly into their absolute memory locations. Virtual stack and heap pointers are maintained, with all new values being added to the end of the appropriate segment. Clearly, it is vital that the data items on the stack are restored in the correct order. Also, due to the linear fashion in which the value table is constructed during the `migrout` phase, the heap must maintain its correct ordering as well.

Pointers also cause problems when placing data values into memory. It is not possible to determine the final value of a pointer until the object it refers to

has been assigned a memory location. Consequently, a table is used to record all pointers, and once all data values have been dealt with, the pointers are converted from their (*Object ID*, *offset*) pairs into machine addresses.

As a last step, the process is restarted by loading the program's binary file into memory, then writing the newly constructed data and stack segments into the address space (using `ptrace`). The preemption point number that represents the continuation address of the process is converted into the correct machine dependent address. Finally, the correct register values are given to the process, and it continues execution.

3.5 The Intermediate Representation

The intermediate file is a machine independent representation of the value table. It lists all data values in a well defined storage format, and if necessary, states the type of the values and the relationship between them. The file format has not been optimized to any great extent.

All data values (`int` and `float`) are encoded using the native storage format for Sun 4 machines. That is, big endian two's complement integers and IEEE floating point values. Since Sun 3 and PowerPC machines also use this format, the Intel 386 is the only machine that needs to perform any format conversion.

The data items are listed in the order: *procedures, global variables, heap values and stack.* This is the order in which they appear within the address space of all architectures currently supported by Tui.

- **Procedures** — The name of each procedure is listed, since it is possible for a pointer to refer to a procedure. No other information is given about the text segment.

- **Global variables** — The variable's name and value are specified. It is necessary to include the name, since variables may appear in a different order on

different architectures. Also, some symbols may exist on one machine, but not on the other; these will typically be machine dependent values and are not normally meaningful to migrate.

- **Heap values** — These do not have names, and the destination machine can not determine the type of the data in advance. Therefore, values are listed alongside their stabs type number. It is necessary that all architectures use a common type numbering system.

- **Stack values** — These are listed within their respective frames. Each frame is identified by the name of the procedure and the number of the call point that created the frame. Parameters, local variables, temporaries and arguments are listed in an order that is consistent among all machines. No variable names are needed.

3.6 Comparison: The Prototype of Tui and the Current Version

The implementation of Tui, as described in the previous sections, is now considered to be complete. However, it is worth discussing the earlier prototype version to show the important discoveries that were made, as well as the tradeoffs between the two different systems.

The original version of Tui used a different approach to scanning the memory of a process. To locate the data stored on the heap, a traversal algorithm (similar to those used in garbage collection systems) was used. While scanning the global and stack data area, any pointers that refer to data in the heap area were followed and the details added to the value table. If the heap data itself contained pointer references, the traversal process continued until all reachable heap data had been located.

Although the prototype algorithm functioned correctly for most programs, there were two major limitations identified. Firstly, it was possible that when migrating non-type-safe programs, a *type conflict* could occur. Secondly, the performance of the value table (when storing information about the data being located) was not satisfactory. These limitations will now be examined in more detail.

3.6.1 Problems Due to Lack of Type-Safety

When a pointer was followed in order to locate a data item in the heap, the base type of that pointer was used to determine the type of that heap data. After much analysis, this approach appeared insufficient given that Tui should function correctly for a non-type-safe language. The following examples will clarify this problem.

Example: Determining Array Sizes

If an array is dynamically allocated on the heap, there is often only a pointer to the beginning of the array. In ANSI C, there is no way of automatically determining its length. As an estimate, the total size of the heap block could be divided by the size of a single array element. However, this is not totally reliable since the programmer may not be using the entire heap block for storing the array.

Example: Type Conflicts

If a pointer refers to a heap block that has already been discovered (by following a previous pointer), both pointers must agree on the data type. If the pointer types differ, there is no way for Tui to ensure that it will correctly interpret the data values.

The following fragment of code is non-migratible since it violates this property. Pointer `a` refers to an integer value (or array of integers) while pointer `b` suggests that this same area of memory stores characters.

```
{
  int *a = malloc(100);
  char *b = (char *)a;
}
```

If this program was migrated, a "migrate-time" error would be reported.

A similar difficulty appears if we vary the ordering in which values are discovered. In the following fragment of code, the pointer `b` refers to an element of the array `a`.

```
{
  int a[10];
  int *b = &a[5];
}
```

If `a` is entered into the value table first, `b` will refer to a known element of the array `a`. On the other hand, if `b` is discovered first, the value table must be carefully rearranged to record that `a` is in fact the most significant data value. This functionality is not impossible to deal with, but the complexity of the necessary code proved to negatively affect its performance.

Solutions

The solution to these two problems is to require the programmer to more carefully specify the type and size of each heap block at their time of creation. In each call to the `malloc` library function, the programmer must use the form:

```
malloc(size * sizeof(type))
```

The C compiler incorporates this size and type into a call to a special version of `malloc` that records the information for later use by Tui.

An alternate possibility would to use the ANSI-C `calloc` procedure, since it requires the programmer to separate the number of elements from the size of each element. However, it is still necessary to use the `sizeof` operator to state the type of the data.

Since this extra information is available, there can be no ambiguity over the type of heap data. A pointer of any type may refer to heap data of any other type, as long as it refers to the beginning of an atomic data value. For example, a character pointer may refer to the first byte of a four byte integer, but not to any of the remaining three bytes.

With the new (current) implementation, "type conflicts" have been reduced to "alignment conflicts", and it is always possible to determine the size of an array.

Aside from the extra cost of storing type information with each heap block, there is a requirement that all calls to `malloc` be put in the correct form. Experience has shown that this is often a simple matter of including a suitable `sizeof` expression, but occasionally more work must be done. For example, the following structure definition may occur:

```
struct foo {
  int len;
  char buf[1];
}
```

The programmers intention is that at run time they will know the length of `buf` and will then be able to allocate appropriately sized storage. However, this violates Tui's rules on using `malloc`. The solution is to rewrite the definition as follows:

```
struct foo {
  int len;
  char *buf;
}
```

With these changes, two calls to `malloc` are required (one for `struct foo` and one for the `buf` array), in each case, the size and type of each object are correctly stored.

3.6.2 Performance Problems

The second limitation of the original garbage collection style algorithm was that due to the potentially random ordering of insertions into the value table, it was not possible to use a linearly expanding data structure. The prototype version used a

splay tree [59] that allows randomly ordered insertions. Although a splay tree will typically give excellent performance for random accesses, there were circumstances where the performance was less than satisfactory.

As an example, when migrating a program that contained a large number of stack frames, each new data item that was added to the value table was guaranteed to be inserted at the end of the table. At the same time, this value was being "splayed" to the root of the splay tree, leading to a very unbalanced structure. It was clear that using the linear table of the revised version of Tui would give better performance for many programs.

The revised algorithm will always give between O(n) and O($nlogn$) performance (based on the number of data items and percentage of those that are pointers), but introduces the restriction that the memory of the process must be scanned in order of increasing (or decreasing) address. This is not a significant restriction since all the architectures supported by Tui have their text, data, heap and stack segments layed out in the same order, although at different memory locations. The introduction of a new architecture may cause a minor problem.

In a final point regarding performance, much effort was made to optimize the Tui algorithm by using a more efficient coding style. These optimizations helped to decrease the execution time of the programs, but did not reduce the "Big-O" complexity of the algorithms.

3.7 Summary

The Tui heterogeneous migration system is able to migrate programs written in ANSI-C on four different architectures, each running a version of the UNIX operating system. The entire system consists of a modified ANSI-C compiler, the `migrout` program for scanning a process and producing an intermediate representation of the data, and finally the `migrin` program for restoring the program on the destination machine. The important details of each component of the system has been described

in detail.

Tui's algorithms have been analyzed and optimized to reduce the time required when migrating programs. The original version of Tui made no assumptions about the source code of the migrating program. However, to improve Tui's performance, and to increase the number of programs that are migratible, a minor restriction was placed on the way that heap data is allocated. This tradeoff has allowed for an algorithm for migrating a process that provides between $O(n)$ and $O(nlogn)$ complexity. Further discussion of Tui's performance will be given in chapter 8.

Chapter 4

Non-migratibility in ANSI-C

One of the major contributions of this thesis is that the concept of migratibility has been defined. In order to do this, a study was performed to determine the non-migratible features that exist within the ANSI-C language. The results of this study provide an understanding of why migratibility problems occur, and how to go about solving these problems.

This chapter will describe several realistic application programs, written in ANSI-C, that a user would be interested in migrating. Next, the set of non-migratible language features that hindered migration of these programs will be described. Chapter 5 uses the experiences of this study to define the concept of migratibility.

4.1 The Migratible Applications

In order to derive a list of non-migratible language features, a representative set of application programs was selected for migration. The source code for each program was obtained, and then a sequence of steps was taken to determine how to migrate the program using Tui. Where necessary, the source code was modified until the program could be successfully migrated. Even though this method was not guar-

anteed to uncover all the possible non-migratible language features of ANSI-C, it helped identify many common problems.

Firstly, details of each of the application programs will be given, followed by a description of the non-migratible language features that were discovered.

4.1.1 Details of the Applications

Each of the programs was written in ANSI-C with the source code being available in the public domain. None of the programs had been written with the intention of using them in conjunction with Tui, however they were representative of the types of programs that users would be interested in migrating. Since all but one of the programs would normally be expected to execute for a long period of time (possibly in the order of hours or days), they would clearly benefit from migration, even if the migration process took several minutes to complete.

Unfortunately, only programs that were capable of executing within the Tui run-time environment were considered. Other software that takes advantage of machine specific features, or of communication with other processes, could not be migrated. Also, commercial software such as word processors or database management systems, would be interesting to study, but the source code was not available.

Following the description of each application, a numerical summary of the important features is given. These figures give an idea of the characteristics of the program, as far as Tui is concerned. The first three values (the number of lines of C code, the number of preemption points, and the number of call points) are characteristics of the program that remain constant, regardless of how the program executes. The remaining values (the number of individual data items, and the percentage of these that are pointers), are simply approximations calculated by migrating the program while it was executing with typical input.

When analyzing these values, there are two important observations to be made. Firstly, several hundred of the program's call points are those that are present

in the run-time libraries. The exact number depends on how many library procedures are being used. Since these libraries do not contain preemption points, that characteristic is not affected in any way.

Secondly, the percentage of data items that are pointers, is an important consideration for the performance of the `migrout` algorithm. If the percentage is negligible, the complexity of Tui will approach O(n), whereas if that percentage increases, the complexity will tend towards O($nlogn$).

The Matrix Manipulation Package

A matrix manipulation package is a very practical piece of software that often requires extensive amounts of memory and CPU time. The particular package that was used [2] is written in ANSI-C, and performs operations such as matrix multiplication and transposition.

Lines of Code	Preemption Points	Call Points	Problem Description	Number of Data Items	% that are Pointers
656	76	323	128 by 128 matrices	121669	almost 0%

The MicroEMACS Editor

MicroEMACS [3] is a cut-down version of the popular EMACS text editor. It contains approximately 20,000 lines of code that were required for compilation under UNIX. Although an editor is not normally a compute intensive application, it is desirable to move the program between different machines in a mobile environment. As an example, a user may wish to migrate a set of desktop applications between their workstation and their laptop.

Lines of Code	Preemption Points	Call Points	Problem Description	Number of Data Items	% that are Pointers
19884	1782	2257	12000 lines of text (an 800Kb file)	1882983	7%

The ST Compiler

This compiler for the ST language [31] "Synchronized Transitions" was chosen due to its complexity. Compilers often contain complex data structures (such as a parse tree) and use language idioms that are not common in other programs. For example, the parsing section of a compiler is often generated by a tool such as "yacc", resulting in code with a unique style.

Lines of Code	Preemption Points	Call Points	Problem Description	Number of Data Items	% that are Pointers
10782	1182	3368	Compiling a small ST program	2224400	14%

The ST Compiler's Output

The ST compiler itself is not a long-lived program, since compilation is normally limited to a few seconds of CPU time. However, the output from the ST compiler (a simulation program written in C), may execute for long periods of time (hours or days) and therefore migration is very desirable.

As part of this research, the ST compiler was modified so that it would produce migratible ANSI-C code. To achieve this, the output from the compiler was migrated as if it were a hand written ANSI-C program. Any non-migratible

language features used by the compiler were noted, and eventually, the compiler was modified so that it would only produce migratible sequences of code.

Lines of Code	Preemption Points	Call Points	Problem Description	Number of Data Items	% that are Pointers
344	97	463	Executing the small ST program	13030	7%

The "bc" Calculator

The "bc" calculator (a standard UNIX command) contains a small language interpreter as well as data structures for storing arbitrary length numbers. It was chosen as being representative of a class of "shell"-type programs that remain active for large periods of time and are therefore important to migrate.

The original intention was to migrate the UNIX Bourne Shell ("sh"), but due to this software's strong dependency on UNIX system calls, this was not feasible. Even though UNIX does not support process migration of any kind, Tui's special file server was used so that a program may continue to access files (using the `read` and `write` system calls) even after it has been migrated. However, this functionality does not exist for system calls such as `fork` and `exec`.

Lines of Code	Preemption Points	Call Points	Problem Description	Number of Data Items	% that are Pointers
8195	655	1188	Executing a 14 line function	18288	7%

Other Programs

A variety of smaller programs were used during the initial construction of Tui. These programs were specially written for testing the migration algorithm, but still provided insight into non-migratible language features. These programs included "factorial", "fibonacci' and "binary tree insertion". Further details are given in chapter 8.

4.2 ANSI-C Non-Migratible Language Features

To determine which non-migratible language features were present in each of the application programs, an attempt was made to migrate the software using Tui. If Tui reported a migration error (either at compile-time or migration-time), the cause of the error was determined, and the source code of the program was modified to remove the non-migratible feature, without changing the semantics of the program. In some cases, extra compiler support was added to automatically identify these non-migratible features.

When a program could be successfully migrated, it was assumed that no more non-migratible features were present. Although complete and time consuming test coverage would be needed to ensure that almost all the non-migratible language features are removed from the program, such testing procedures were not used. Appendix B describes the simpler approach that has been taken.

A second method used to determine the non-migratible language features was to examine the official description of the language [39] and attempt to predict which of the language features would cause problems. Given that the language description was not written with migration in mind, some of the language's semantics are quite vague or ambiguous.

A general description of each problem will now be given. As a prelude, it is interesting to note that each of the non-migratibility problems can be classified into

one of the following general classes. Each class describes the overall reason why the problem occurs.

1. *Conflicting Type Information* – A problem of this type will lead Tui to believe that a particular memory location has more that one data type associated with it. Therefore, Tui will not know how to correctly translate the data value.

2. *Lack of Type Information* – This occurs when a data value that is required by the program is not correctly identified and is therefore not migrated. This is usually because the compiler did not pass enough type or location information to the migration tool.

3. *False identification* – When Tui attempts to migrate a data value that is not actually part of the program. This occurs when Tui is supplied with incorrect information for locating data values.

4. *Incorrect Conversion of Values* – When data values are successfully located and typed, but are not converted correctly for use on the target machine. This is either because the data type has different properties, or the value itself is not accurately representable.

The assumptions to be made when considering the specific problems are that the migratible program:

- Is written in ANSI-C.
- Possibly contains compiler warnings, but not errors.
- Does not contain any operating system specific functions, other than those supported by the Tui file server.
- Is being migrated by the original version of Tui (using a garbage collection style algorithm to locate data values). This version makes no assumptions about the source code of the program.

Section 5.3 will discuss how these problems can be solved using the framework provided in chapter 5.

1. Conflicting Type Information

These problems all occur when a memory location has more than one type associated with it.

- *1.1 Union data type* – The programmer decides that two or more data values can be stored at a single memory location since those values will never be used concurrently. It is entirely the programmer's responsibility to ensure that this property holds. A migration tool cannot correctly translate a union variable since it does not know which data type is currently active.

  Some programs take advantage of the dual storage and use the union data type as a means of manipulating the underlying representations of the values. For example, it can be used to determine whether the machine stores integers in little or big endian format. This can be very useful, but often leads to non-portable code. The use of the union data type can be detected at compile time, however it is, in general, only possible to detect non-portable use at run-time.

- *1.2 Pointer casting* – Pointers are normally assigned values by taking the address of a variable, by calling a function that returns a pointer (e.g. `malloc`), or by fetching the value of a second pointer. In any of these cases, if the type of the source and destination values are different, then the programmer is creating a conflicting indication of what the pointer is referring to, therefore confusing the migration tool.

  Even though it is always possible to warn the programmer that they are using a pointer cast in their program, it is not possible to always determine whether a cast will lead to a type conflict.

- *1.3 Mismatched parameters* – When a function is called with arguments, it is desirable that the type of each argument be consistent with the type of the corresponding parameter. Most compilers will warn the programmer if any differences are noticed. If these warnings are ignored, the program will implicitly need to convert the arguments from one type to the other, possibly converting between different pointer types, or from an integral value to a pointer. In these situations, the migration tool will not know how to correctly translate the data.

  In the worst case, the programmer may wish to call a function that has not been declared in any way, either with a definition or a prototype. The compiler will have no idea of the expected data types and is therefore unable to give an accurate warning about type mismatches.

  Type mismatches can be detected at compile time, but may or may not cause problems for the migration tool, depending on the data types involved.

- *1.4 Reused variable storage* – When the compiler is deciding upon how to store a local variable or temporary value, it chooses between allocating space on the stack, or using a register. A common optimization is to reuse a storage location when it is known that two or more variables will have mutually exclusive lifetimes (such as with the union data type). If this reuse is not properly described by the compiler, the migration tool will not know which data value is currently active.

- *1.5 Lack of Type Checking because of separate compilation* – Large programs are frequently divided into many separate source files, each being compiled individually to create separate object files. The last step is to link all object files together to create an executable program. In ANSI-C, any variable that is defined in one file, but used in another, has the potential of causing a type conflict if the programmer does not supply exactly the same variable

declaration in both of the files.

2. Lack Of Type Information

These problems occur when the compiler has not produced, or is not able to produce, enough type information to describe the data being migrated.

- *2.1 Generic (void) pointers* – A pointer with base type `void` is able to refer to an object of any type, without receiving a compiler warning. This feature is useful when storing an arbitrary object within generic data structures such as a linked list or binary tree package. From the migration tool's point of view, the object pointed to by a generic pointer does not have a type, and is therefore unable to correctly translate the object unless another (typed) reference to the object is available.

- *2.2 Untyped Dynamic Allocation* – When the `malloc` function is used to dynamically allocate storage, the programmer requests a memory block based on the number of bytes required. The way in which this block is used is left for the programmer to decide. The generic pointer returned by `malloc` can be cast to a pointer of any type. In most cases, the memory block will be used to store a single object (e.g. a C structure) or an array of objects. In some exceptional cases the block may contain a variety of data types.

  Given that the compiler can not be totally aware of how each memory block will be used, the migration tool is not able to correctly translate the data values. In particular, it will not be aware of how many array elements are present within each block. The revised version of Tui (described in section 3.6) has been able to partially solve this problem.

- *2.3 Untyped Internal Constants* – Most compilers that generate type information for use by a debugger or migration tool will only be concerned about variables that were explicitly created by the programmer. Often the compiler

will generate further data values (such as floating point and string constants) that are not visible to the programmer. If the compiler does not declare these values to the migration tool, they will not be translated.

- *2.4 Variable Length Argument Lists* – For some C functions (such as `printf` or `execl`), it is important that the number of arguments that can be passed to the function be flexible. At runtime, the function itself is responsible for determining the correct number and type of arguments that were passed to it, using a technique such as examining a format string.

  Since, in general, there is no way for the function to determine whether the arguments were of the type it was expecting, the problem is similar to calling the function without having seen a correct function prototype.

- *2.5 Command line argument arrays (argv)* – The size of the `argv` array varies depending on the number of command line arguments passed to the program. Since ANSI-C does not support the ability to determine the size of a dynamically created array, the usual way to determine the correct size is to examine the `argc` variable.

  Given that `argv` can not be cleanly described by the language's type system, the basic migration tool is unable to determine the size of `argv` or any other variables that use this type of data structure.

- *2.6 Setjmp and Longjmp* – These functions allow a program to perform a non-local goto command. That is, the `setjmp` function will record the current execution location and stack pointer so that at a later time, the `longjmp` procedure can return to that point.

  The context information is stored in a structure known as a `jmp_buf` that typically contains two generic pointers (one into the code, and the other into the stack). Since the pointers do not refer to objects that are normally identified by the compiler, the migration tool is unable to translate the `jmp_buf`

structure.

3. False Identification

These problems occur when the migration tool incorrectly believes in the existence of data.

- *3.1 Casting Integral Values to Create Pointers* – Since pointers are used to identify blocks of memory, casting an integral value to a pointer will often give a false indication of where data can be found.

  There are two common reasons for wishing to perform this type of cast. Firstly, if the program uses a hardware device, hard-coding a memory location into the source code is a common technique. However, it is very non-portable and non-migratible due to hardware dependencies.

  Secondly, a programmer may attempt to store an integer value inside a pointer variable, simply because it was a convenient thing to do. As an example, the ANSI-C `signal` function expects to be given a pointer to a function to call when a signal occurs, however passing the integer 1 is the conventional way of asking that the signal be ignored.

  One exception to this problem is the use of 0 to represent the NULL pointer, as Tui is fully aware of this convention.

- *3.2 Pointers Referring to Illegal Memory Locations* – The ANSI-C standard states in A7.7 that pointer arithmetic may only be performed in such a way that the new pointer refers to another element of the same array, or the first memory address beyond the end of the array, otherwise the result is undefined. In the case where a pointer refers to memory outside its intended array, either it will refer to an empty block of memory or a block that was allocated differently. In either case, the migration tool will receive incorrect information.

There are several cases where having undefined pointers is desirable. As an example, arrays that do not have a zero-based index could benefit from a "virtual base pointer" to a location before the start of the array. When finding a particular element's memory location, there is no need to subtract the array's starting index before adding the base memory location, as this has already been taken into account when calculating the virtual base pointer.

In most cases, these problems cannot be detected until migration-time or run-time, although a large amount of compile-time analysis could suggest that illegal pointers would be used.

- *3.3 Dangling pointers to deallocated memory locations* – Pointers can also be made to refer to illegal memory addresses by deallocating memory, rather than by performing illegal arithmetic. The first case is when a pointer refers to a stack based object, and the stack frame is deallocated before the pointer is destroyed. Secondly, a pointer may refer to a heap block that has been deallocated by a call to `free`.

  In both of these situations, it is likely that the pointers will never be dereferenced by the program, but the migration tool will still use the pointer to locate information. If the memory has been reallocated since the pointer was created, conflicting type information will result.

- *3.4 Uninitialized pointers* – Before an automatic (local) pointer variable is assigned to for the first time, its value is undefined (ANSI-C A8.7). If the program is migrated before the first assignment, the pointer may give misleading type information to the migration tool.

- *3.5 Omitted return values* – As with uninitialized pointers, if a function is expected to return a legal pointer value but the return expression is omitted, then an undefined value will be returned. Although omitting a return value is considered to be a programming error, it is still possible to construct and

therefore migrate this type of code.

4. Incorrect Conversion of Values

These problems arise when data values are converted between the formats of the source and destination machines.

- *4.1 The "sizeof" Operator* – This operator returns the (machine dependent) number of bytes required to store a variable of a particular data type. If this numeric value is assigned to a variable, or passed to a function, the value will lose its significance after migration to a machine with different sizes. Translating values that are derived from the result of a `sizeof` operation is not generally possible.

- *4.2 Double Meaning of "char"* – The character data type is frequently used for two conflicting purposes. Firstly, it can be used to store a character value, and secondly it can store an 8-bit number. During migration to a machine that uses a different character set, there is no way to determine whether the values should be remapped as a character or as a number.

- *4.3 Format Conversion Losses* – Different machines have different data type representations. When migrating from one processor to another, the data values may lose accuracy or the properties of the data type may change. This may not cause the migration tool to fail, but can result in an incorrect program.

  For example, one machine may use the IEEE 64-bit floating point system to represent the ANSI-C `double` type, whereas another uses a proprietary system. Some programs that use floating point arithmetic choose to calculate a value for ϵ, which is the smallest step between two consecutive floating point values. Although this value can be correctly computed on all architectures, the exact value depends on the floating point representation and therefore may be incorrect after migration.

A second example would be when migrating to a processor that has a different integer size. If a 64-bit integer was translated to a 32-bit machine, the target machine would not be able to hold a large value. In the reverse direction, the 64-bit machine could always hold a 32-bit value, but the numeric properties of arithmetic overflow would be different. Further elaboration on this issue will be given in section 5.4.

In the next chapter, these non-migratible language features of ANSI-C will be used to formally define the concept of *migratibility.*

Chapter 5

Defining Migratibility

In order to fully understand what it means for a program to be migratible, it is important to clearly and concisely define *migratibility* in a general sense, without being too concerned with particular language features. A general definition will not only present an overall illustration of the issues involved, but will help explain the solutions to non-migratibility presented in this thesis.

Up until now, existing heterogeneous migration systems have all either assumed that a program must be *type-safe* for it to be migratible, or have avoided this issue altogether. No effort has been made to determine the true relationship between the two properties. For example, in the Colorado Springs system [56], the following statement is made:

> *We have restricted our consideration to the equivalent of a strongly typed language with no variant records.*

In the Emerald mobility system [62], no mention is made of this type of restriction, but since the Emerald programming language is already type-safe, the issue did not arise.

The research presented in this chapter will demonstrate that although the assumption of type-safety is usually valid, the set of programs that are type-safe is

not the same as the set that are migratible. The two concepts will be defined, and this will be followed by a discussion of how the approach taken to achieve type-safety can be used and extended to achieve migratibility.

5.1 The Definition of Type-Safety

The concept of *type-safety* was brought about by the observation that a large number of programming bugs occur when a programmer misuses data values. For example, if a variable is declared to be of one type, but is interpreted as another type, the result is often undefined. Also, if the program accesses data that has not officially been declared (such as with an array bound violation), it will result in the corruption of data values. In a type-safe language, the programmer is not permitted to use these undesirable features, therefore reducing the likelihood that a program contains bugs.

It has long been recognized that type-safety is an important property, as it has been incorporated into many modern programming languages (such as Emerald [52], Java [30] and Ada [1]). Even though it is often mentioned as a well-known concept, it is still possible to describe type-safety in a variety of different ways. Sethi [55] defines type-safety as:

> *Type checking ensures that the operations in a program are applied properly. [...] A program that executes without type errors is said to be type-safe*

Ghezzi and Jazayeri [28] suggest that the use of strongly-typed languages will help a program become type-safe, although there still exist some circumstances, such as array bound checking, where run-time checks are required to ensure type-safety.

A more detailed definition, followed in this thesis, requires the definition of four abstract operations that are applied to an *area of memory*. That is, a collection of consecutive bytes can be allocated, deallocated, written to, or read

from. A particular set of consecutive bytes may be used for a variety of purposes over the lifetime of the program.

The abstract operations, and the associated rules are listed below. Any program that obeys these rules of data access is guaranteed to be *type-safe.*

- **Allocate** – Associate a data type with an area of memory. None of the bytes in this area of memory can be allocated again until they are firstly deallocated. That is, successive **Allocate** operations on an area of memory must be separated by **Deallocate** operations.

- **Deallocate** – Return an area of memory to the pool of available bytes. This area of memory can not be used until it is allocated again. Also, there must not have been any previous **Deallocate** operations on the area of memory, unless followed by an **Allocate** operation.

- **Write** – Write a value with a given type to an area of memory. The data type of the value must be consistent with the type associated with the block by the most recent **Allocate** operation. There must not be any **Deallocate** operations that are more recent than the **Allocate**.

- **Read** – Read a value of a given type from an area of memory. The type of the value must be consistent with the value of the most recent **Write** operation on that area of memory. There must not be any **Allocate** or **Deallocate** operations on that area of memory since the most recent **Write** operation.

In addition to this definition, it is important to realize that a *type-safe language* is not the same as a *type-safe program.* A language is type-safe when it does not contain any language features that permit the programmer to violate the type system. Most type-safe compilers take the approach of allowing the use of some non-type-safe features, as long as misuse can always be reported at run-time. For example, the array data type is a potential candidate for a type violation, but as long as run-time checks are able to report array bound errors, no violation could actually occur.

On the other hand, a type-safe program can be written in a non-type-safe language, as long as it does not make use of any non-type-safe features. In the general case, it is not possible to determine whether a program is type-safe, but careful analysis of the program code can detect certain problems, or suggest where potential violations may occur. Although this solution can be very helpful for reducing program bugs, it is only a heuristic in comparison to using a type-safe language.

5.1.1 Contributing Factors

When designing a type-safe language or writing a type-safe program in a non-type-safe language, there are several factors that must be taken into account. Each of these factors is commonly used as a means of gaining type-safety, although not all of them can guarantee success.

1. **Language Restriction** – In most modern type-safe languages, the desire is to create an entirely new language, rather than remain compatible with an existing specification. In this situation, any language features that are non-type-safe can easily be removed from the language, or be restricted in functionality.

2. **Static Analysis** – To determine whether a program is type-safe, the program source code can be thoroughly analyzed to detect safety problems. In some cases, such as in the use of pointer type casting, analysis can always detect violations. However, in situations such as array bound violations, it is impossible to detect all possible problems without actually executing the program. Although this technique will not guarantee type-safety, tools such as "lint" (a standard UNIX tool [70]) are often beneficial in suggesting potential problems.

3. **Run-time Code Checks** – If it is not possible to determine whether a program is type-safe at compile time, an alternative solution is to add run-time code. Arrays are considered a very important language feature that cannot

be statically checked, so run-time code must be used to ensure type-safety. If a safety violation occurs, either the run-time system will report an error, or an exception will be thrown. In either case, common usage demonstrates that these run-time checks are accepted by programmers as a valid way of ensuring type-safety.

An extreme case of run-time checking is demonstrated in the "purify" [18] software package that performs run-time checking of pointer use. Given a set of object files, purify adds run-time checks which verify that pointers and heap blocks are being used legally. Although this tool reduces the program's efficiency, this software is used during development as an *ad hoc* method of verifying the program's correctness.

In summary, the methods used to obtain type-safety (therefore reducing bugs) have been a combination of the three approaches listed above. The actual solution used in any specific implementation will depend on the assumptions the designer wishes to make. That is, for each non-type-safe language feature, the most suitable approach will be chosen.

One of the most important features of type-safety, with respect to this discussion, is that it is an absolute property of the source code. Although static analysis and run-time checks can be used to inform the programmer of type-safety violations, they do not affect the safety of the program. To clarify, there are no circumstances where adding extra run-time code or performing extra static analysis can stop the program from attempting to violate the type system, without changing the semantics of the program. In any situation, the programmer must eventually modify the source code to ensure type-safety.

5.2 Deriving a Definition for Migratibility

The purpose of this research has been to identify language features that are non-migratible. Having done this, it is now important to classify these problems and present solutions in a systematic manner. After experimentally discovering the problems in ANSI-C, it was seen as important to define migratibility with the same approach that has already been taken with type-safety.

For formal purposes, migratibility is defined as:

> *A program is migratible, with respect to two language implementations, A and B, if and only if the input and output of the program is always the same (bit for bit), regardless of whether it is executed on implementation A or implementation B, or whether it is moved from implementation A to implementation B at an arbitrary preemption point. Preemption points must be placed at frequent intervals throughout the program such that the delay introduced by migration is less than the time delay parameter, τ.*

In practice, this definition could be relaxed to allow a certain number of trivial differences. It is usually acceptable if implementation A and implementation B use different character sets, as long as the output appears to be the same when presented to the user. Likewise, the rounding policy in floating point calculations may be different (that is, in the number of significant numbers, or the accuracy of those digits), as long as the user does not perceive a significant difference.

The time delay parameter, τ, will only be briefly mentioned during this formal definition of migratibility. Elsewhere, τ is implicitly defined to be an amount of time that is acceptable to the user of the migrating program. The actual value will differ, depending on each user's expectation, and may vary from a small number of seconds for interactive programs, to several minutes or hours for batch jobs. The primary concern is that migration of a process must provide better performance as

perceived by the user. It is not acceptable to degrade the program's execution time.

If we fail to define τ, it would be possible to say that any program is migratible. For example, migrating a program at its point of termination is guaranteed to succeed since there is no longer any valid state to be transferred. However, a program may require an unacceptable amount of time in order to reach the terminating point of execution.

5.2.1 Contributing Factors

As with type-safety, a series of approaches can be taken to solve the non-migratibility problems, and the designer of a migration system must choose between the alternatives. Firstly, the approaches taken by type-safe systems will be listed, followed by the new approaches that are only relevant for migration.

1. **Language Restriction** – As with type-safety, restricting the language to not allow the use of certain language features is an excellent technique for achieving migratibility. However, since we are primarily concerned with migrating programs written in traditional languages, restricting the capabilities of the language is very undesirable, and should not be done unless totally unavoidable. However, as programming languages develop over time, and become more and more type-safe, this approach will most likely be increasingly used.

2. **Static Analysis** – Techniques can be used to detect use of non-migratible features, and to suggest source code changes that the programmer should make. As with type-safety, many of these techniques can only be heuristics, and there is no guarantee that migratibility will always be achieved.

3. **Run-time Code Checks** – To catch the use of non-migratible language features that couldn't be detected statically, run-time checks are important. These checks do not avoid migratibility violations, but they should give an accurate indication of how the program should be altered.

4. **Run-time Code Support** – With migration, the potential exists for adding run-time code that can make a program migratible, independently of the contents of the source program. For example, by ensuring that uninitialized pointers are set to NULL, we can remove one migratibility problem.

5. **Amount of Type Information** – The migration tool depends heavily on type information provided by the compiler. Some migratibility problems can be solved by simply acquiring and maintaining more type information. The cost of this method is in the time and space required to generate and store the information. This approach is already taken in dynamically typed languages such as Scheme.

6. **Functionality of the Migration Tool** – The migration tool depends heavily on the way in which a program is stored in memory. At a migration-time cost, special functionality could be added to deal with some of the non-migratible problems, either by performing migrate-time checks, or by executing algorithms that can deal with special cases.

   In a similar way that run-time checks are used to ensure type-safety, migration-time checks could be used to report an error, throw an exception, or abort the migration. Aborting a migration would be most appropriate if we are able to continue the process (remaining on the source machine) as if migration had never been attempted, possibly trying again at a different point in time.

7. **Machine Similarities** – For some language features, knowing whether they will be non-migratible depends on the differences between the source and destination machines. The higher the number of characteristics (such as data formats, data locations or operating system type) that are known to be different, the higher the number of non-migratible features. Therefore, by restricting the machine differences, migratibility is more easily obtained.

5.2.2 Formalizing Migratibility

As with the approach taken to obtaining type-safety, migratibility can be obtained by approaching each undesirable language feature and solving it with one of the solutions given above. Clearly there are a range of tradeoffs to be made, so the final solution will depend on the language, environment, and costs that the user is prepared to suffer. As new migration systems are designed and constructed, these factors should be taken seriously.

To formalize this idea, if we imagine a 7-dimensional space, with each of the preceding factors assigned to an axis, there would exist a set of points in that space that represent *useful migration systems.* Any migration system that matches a valid point in the space will be guaranteed to always migrate programs correctly or to somehow refuse migration of non-migratible programs. Determining this exact set of points is not within the scope of this thesis.

Three migration systems will now be described by showing their approximate position within this 7-dimensional space. The importance of each factor to that system will be stated with a ranking of LOW, MEDIUM or HIGH. The details for the following systems are summarized in the table.

1. Tui – The focus of Tui has been to restrict the language as little as possible by providing as many automatic methods of obtaining migration. Since this was not entirely possible without seriously sacrificing run-time performance, a few restrictions were placed on the language design and on the machine differences.

2. The Colorado Springs system [56] – In this system, it is assumed that the language must be type-safe, and that the machines must have very similar properties. As a result, the run-time overhead and complexity of the migration tool is kept to a minimum.

3. An interpreter – In this (fictitious) case, the destination machine is required to simulate the processor of the source machine. Although it can obviously

be argued that this is no longer heterogeneous process migration, it is essentially the type of system that is used in the Java virtual machine [42] to provide a consistent platform across a wide variety of architectures. Also, it demonstrates the limits to which this migration framework could be used.

The details are as follows:

	Tui	Colorado-Springs	Interpreter
Language Restriction	MED	HIGH	LOW
Static analysis	MED	LOW	LOW
Run-time checks	LOW	LOW	LOW
Run-time support	MED	LOW	HIGH
Type-information	MED	MED	LOW
Migration tool	HIGH	MED	LOW
Machine similarities	LOW	HIGH	LOW

5.3 Solving each of the ANSI-C migratibility problems

To demonstrate that the preceding migratibility framework is useful for creating a migration tool, we now present an approach taken to solving each of the non-migratibility problems listed in section 4.2. A description of how the current version of Tui solves the problems will be followed by the proposal of an alternate (unimplemented) solution. For each of the problems, there will be one or more factors given (e.g. run-time performance, migration-tool complexity) that is impacted.

It must be noted that when constructing Tui, the intention was to determine which language features were non-migratible. As a consequence, migratibility was initially achieved by the static analysis approach. After identifying which language

features would cause a problem, the compiler was instrumented to warn the programmer about potential pitfalls, regardless of whether or not the problem would actually stop the program from being migratible (as is done with the "lint" tool). Although a small number of problems were solved by adding more type information, changing the run-time code, and adding features to the migration tool, this was not the main intent.

The proposed solution (at a different point in the 7-dimensional space) is aimed at those users who are not concerned about run-time performance, but instead have a strong desire to migrate code without making any attempt to analyze code for themselves. The intention is that the compiler, run-time system, or migration tool will inform them of exactly where any migration problems have occurred, in a similar way that a run-time check would state where a type violation had occurred. After the program has been tested for migratibility, the complex run-time checks could be removed in order to gain speed (as is done with the purify tool).

It is very important to realize that these solutions are only two of many possible approaches. The proposed system takes the approach of providing *run-time checks* that inform the user of where non-migratible feature are being used. In many cases, the same checks would be performed if we were attempting to achieve type-safety. Other proposed systems (at other points in the 7-dimensional space) might take the approach of providing *run-time support* to silently migrate programs without user intervention, or may be more strict about the *machine similarities.* The focus and design of any migration system depends on the requirements of the user.

1. Conflicting Type Information

- *1.1 Union data type*

  - Tui : *Static Analysis* – The compiler will report any usage of the union data type, and will then convert it to a struct data type. The programmer

must determine whether the data value will now act as expected.

- Proposed : *Run-time Check* – Maintain a run-time type tag that records which field was most recently written to. If the program attempts to read from a different field, a run-time error will be reported. This solution will disallow programs that intentionally misuse union values, or try to predict a program's memory layout.

- *1.2 Pointer casting*

  - Tui : *Static Analysis* – The compiler will warn the programmer about all pointer casts in the program, regardless of whether they will cause problems or not. It is the programmer's responsibility to determine that the pointers are not misused. That is, it is legal to perform a pointer cast, but it is not legal to dereference the pointer and potentially use data in an incorrect way.

  - Proposed: *Run-time Check* – To guarantee that data is not accessed incorrectly, it is possible to tag every data value with a type-tag (as in dynamically typed languages such as Scheme). Each time a pointer is dereferenced, the data is checked to ensure that it is of the correct type. Although this dramatically increases execution time and memory requirements, it provides a foolproof means of detecting whether data is being corrupted, without limiting the language.

- *1.3 Mismatched parameters*

  - Tui : *Static Analysis* – The compiler requires that prototypes must be supplied for all functions and variables that are accessed before being declared. If a prototype is missing, a warning is given at compile-time.

  - Proposed : *Static Analysis* – Since supplying prototypes is an accepted part of ANSI-C, the approach taken by Tui is the best solution.

- *1.4 Reused variable storage*

  - Tui : *Run-time Support* – The code generator must ensure that registers and stack locations are only used for a single purpose within each procedure.
  - Proposed : *Type Information* – The use of a more modern type of debugging information [4], rather than the aging 'stabs' system would allow reuse of storage locations.

- *1.5 Lack of Type Checking because of separate compilation*

  - Tui : *Static Analysis* – Tui performs extra type checking after the standard linker has been applied.
  - Proposed : *Static Analysis* – Tui's solution is satisfactory since it ensures that variables are declared in a consistent manner.

2. Lack Of Type Information

- *2.1 Generic (void) pointers*

  - Tui : *Migration Tool* – In the current version of Tui, pointers are not used to locate data. This is because type information is determined either at compile time, or at the point at which memory is dynamically allocated. Any void pointer can be correctly migrated, as long as it refers to a known data value.
  - Proposed : *Migration Tool* – The solution used by Tui is acceptable since the `migrout` algorithm does not rely on knowing the type of a pointer in order to translate the data it refers to.

- *2.2 Untyped Dynamic Allocation*

- Tui : *Language Restriction, Static Analysis, Run-time Support, Migration Tool* – The most significant language restriction required by Tui is that all calls to memory allocation functions (such as `malloc` and `realloc`) must be tagged with the type of data that the memory block will store. The compiler must ensure that the programmer uses the `sizeof` operator within a memory allocation call. For example:

  ```
  ptr = malloc(sizeof(int) * 100)
  ```

  This specifies that the block will contain an array of 100 integers. The migration tool can now correctly determine the size of the block, translate the data, and verify any pointers that refer to the block. Experience with the example application programs (see section 4.1) showed that minimal modifications were necessary.

  A limitation is that writing a new version of `malloc` is not possible. It is important that the compiler is able to associate a unique type with each allocation call. Therefore, modifying the existing functions, or writing an entirely new set of allocation procedures will not be possible.

- Proposed : *Language Restriction, Static Analysis, Run-time Support, Migration Tool* – Only a minor modification from Tui would be needed to add more flexibility. Instead of only allowing a single data type per heap block, the `malloc` call would be allowed to contain many data types. For example:

  ```
  malloc(100 * sizeof(int) + n * sizeof(char))
  ```

  Even though this extension will work for most programs, some users will still wish to violate this type description. Such programs will not be migratible.

- *2.3 Untyped Internal Constants*

- Tui : *Type Information* – For every data value that the compiler generates, the correct type information is generated.
- Proposed : *Type Information* – The method used by Tui is satisfactory since it completely solves the problem.

- *2.4 Variable Length Argument Lists*
  - Tui : *Static Analysis* – The compiler warns of potential problems and requires the user to verify that the arguments and expected parameters will match.
  - Proposed : *Run-time Check* – When a procedure accepts a variable number of arguments, the code generated for making the call must also pass a type-tag for each value. The compiler will generate run-time code so that the callee will check this information (for example, within `va_arg`) and report a run-time error if violations are found.

- *2.5 Command line argument arrays (argv)*
  - Tui : *Migration Tool* – The migration tool understands how to determine the length of `argv`, so values in the array can always be migrated. However, other arrays that are similar to `argv` will not be noticed by the migration tool.
  - Proposed : *Migration Tool* – Tui's solution is satisfactory.

- *2.6 Setjmp and Longjmp*
  - Tui : *Static Analysis* – The compiler warns the user of their use, but does not attempt to fix the problem.
  - Proposed : *Type Information, Migration Tool* – By adding extra information about the location of `setjmp` calls, correctly migrating the `jmp_buf`

data structure would be possible. This solution requires a minimal cost at migration time, and no extra cost at run-time, although a moderate amount of extra type information must be generated by the compiler for use by the migration tool. This mechanism could be extended to allow migration of a user level threads package, as long as threads could only be suspended at well known locations in the program code.

3. False Identification

- *3.1 Casting Integral Values to Create Pointers*
  - Tui : *Language Restriction* – This is definitely not allowed in the language. Apart from a few small examples (e.g. the `signal` function), performing this type of conversion is machine-dependent.
  - Proposed : *Language Restriction* – Tui's solution is satisfactory. Even though it doesn't guarantee migratibility for all programs, it is difficult to provide a better solution.

- *3.2 Pointers Referring to Illegal Memory Locations*
  - Tui : *Migrate Tool* – If the pointer refers to a memory location that is not known to be valid, a warning is given at migration time, and the pointer value is set to NULL. Since the pointer is undefined anyway, it should not be used again, so setting it to NULL will help track misuse of the value. If the pointer refers to a valid area of memory that is outside its original heap block (or stack based variable), then it is not possible to detect that the pointer is illegal.
  - Proposed : *Run-time Check* – To ensure that a pointer will never move out of range, each pointer could be tagged with a descriptor that states the range of valid values. Whenever a pointer value is created (i.e by

taking the address of an object, or by fetching a return value from a function), the descriptor is created, and whenever pointer arithmetic is performed, a run-time range check will be used to verify that the pointer is being used correctly.

- *3.3 Dangling pointers to deallocated memory locations*

  - Tui : *Migration Tool* – When an object on the heap or stack is deallocated, any pointers referring to those memory locations will become undefined. At migration time, any pointers that refer to unallocated blocks will be set to NULL since it should never be used again. However, if the memory location is reallocated and then the pointer is dereferenced, the program will continue executing and migration will occur, but in both cases, the result may or may not be correct.
  - Proposed : *Run-time Check* – The same technique that was used for detecting illegal pointer values can be used to detect accesses to memory that has been reallocated. If a sequence number is associated with every heap block or stack frame, a pointer value can be verified before being used to access data. If the memory area's sequence number does not match that of the pointer, then a run-time error will occur.

- *3.4 Uninitialized pointers*

  - Tui : *Run-Time Support* – The compiler must ensure that all local pointer variables have extra run-time code for initializing them to NULL. Also, all heap blocks must be initialized to zero upon allocation. A small amount of run-time overhead, typically requiring one or two instructions per pointer initialization.
  - Proposed : *Run-time Support* – Tui's solution is satisfactory as it ensures that no pointer can be created with a non-NULL value.

- *3.5 Omitted return values*

  - Tui : *Static Analysis* – The compiler will warn the user about the missing return statement.

  - Proposed : *Static Analysis* – Tui's solution is satisfactory.

4. Incorrect Conversion of Values

- *4.1 The "sizeof" Operator*

  - Tui : *Language Restriction* – The "sizeof" operator can only be used inside a call to the `malloc` function. All other occurrences must be removed from the program.

  - Proposed : *Machine Similarities* – Since all of the architectures supported by Tui have the same data type sizes, it is reasonable to allow the sizeof operator to be used. No matter how the sizeof value is used, we can be sure that the result will be correctly migrated.

- *4.2 Double Meaning of "char"*

  - Tui : *Machine Similarities* – Tui assumes that both machines use the same character set.

  - Proposed : *Machine Similarities* – Tui's solution is satisfactory since almost all modern machines use the ASCII character set, or a compatible derivative.

- *4.3 Format Conversion Losses*

  - Tui : *Machine Similarities* – Tui assumes that data types will be similar enough so that conversion losses will not occur.

- Proposed : *Machine Similarities* – Tui's solution would probably be satisfactory, although conversion of large integers from 64-bit machines should result in a migration-time error.

5.4 Type-safety and Migratibility Are Not the Same

As previously discussed, type-safety and migratibility have different purposes. Type-safety is important so we can limit the possibility of bugs, whereas migratibility means that a program can move between two machines. To demonstrate that these concepts are different, I now present examples that fall into one category or the other, but not both.

5.4.1 Type-Safe Programs Are Not Always Migratible

The definition of migratibility requires that the program produces identical output on both machines, whether or not migration occurs during the program's execution. This requirement does not restrict the use of machine dependent values to tailor the program at run-time. The following examples demonstrate this.

Figure 5.1 is an example of a hash function that takes an array of 5 unsigned integers as the key and returns an index position within the hash table. This position is calculated by summing the 5 integers and taking the remainder of that sum when divided by the table size. Assuming that the input parameter is correct (checks have been omitted from this code), this procedure will be type-safe. Since the modulus operator is applied at each step, we can also be sure that integer overflow will not occur.

Even though type-safety is achieved, the program will be non-migratible if transferred to a machine with a different integer size. Since the size of the hash table is dependent on the maximum integer value of the underlying machine, the hash values for a given key will be different after migration has taken place. However, it is important to note that this procedure will function correctly if executed solely

```
#include <limits.h>

#define TABLE_SIZE (INT_MAX % 1000)

int hash(unsigned int *key)
{
  int n;
  unsigned int t = 0;

  for (n = 0; n != 5; n++){
    t = t + key[n];
    t = t % TABLE_SIZE;
  }

  return t;
}
```

Figure 5.1: A Type-Safe, Yet Non-Migratible Sample of Code

on either the source or destination machine. Since the hash values are internal to the program, they will affect the storage layout of the program's data, but they do not affect the output.

A different example is seen in ANSI-C where the "sizeof" operator is used to determine the number of bytes required to store a structure. This size can be stored in a variable and will later be used to allocate memory of the correct size for storing the structure. The program is type-safe, but if migration occurs, the value of the variable may be incorrect for future memory allocation.

In general, the following type-safe language features are non-migratible:

- Since a type-safe program is only intended to run correctly on one machine at a time, it can freely make use of machine dependent values such as those returned by "sizeof". A migratible program cannot use these values as they will become meaningless after migration.

- For the same reason, a loss of data precision can occur when a program is migrated. For example, when moving from a machine with 64 bit integers, to one that has 32 bit integers. The program may still be type-safe, but will not continue correct execution if integer values are too large. It is important to note that this type of program would not be type-safe if it depended on the overflow properties of the integer data type. That is, programs relying on the ability to store very large numbers, would not work on a 32-bit machine, so migration is automatically out of consideration.

- A compiler for a type-safe language does not have any limitations on the code it produces, therefore it may contain many optimizations that are non-migratible. For example, a virtual array base is a technique used to reduce the amount of computation required for each array access. However, this leads to the use of a pointer that may refer to an illegal memory location.

- Unless modifications are made to a type-safe language's compiler, it will not report necessary migration information such as preemption points, call points and stack frame sizes.

5.4.2 Migratible Programs Are Not Always Type-safe

The requirements of migration only dictate that the program must produce the same results, whether executed on machine A or machine B, or whether it is migrated during execution. If the program were to consistently violate the type system (such as with a memory access violation) on both machines, it is important to ensure that exactly the same violation occurs after migration has taken place. This type of program is clearly migratible, but it is not type-safe.

Figure 5.2 demonstrates a similar idea in that a non-type-safe action is performed, yet the program will be migratible. In this example, the procedure is called with a pointer to a character, but the body of the procedure contains a pointer cast and an assignment that stores the value into an integer pointer. The type cast

```
void casting(char *char_p)
{
  int *int_p;

  int_p = (int *)char_p;
}
```

Figure 5.2: A Migratible, Yet Non-Type-Safe Sample of Code

is clearly violating the type-safety of the program, but since the integer pointer is never accessed, and Tui does not use pointers to determine the type of data, this program will be migratible.

The following more specific examples are migratible, but not type-safe:

- Type-safe languages must be concerned about restricting misuse of all types, whereas to migrate a program, we do not need to be so strict. For example, in some languages it is possible to create subranges of integers. It is non-type-safe if you are permitted to create a value outside the specified range, but for migration we only care about maintaining the value (whether in range or not) across different platforms.

- Migratibility is concerned with the state of the program at the point of migration. Non-migratible features can still be used, as long as their effects are confined to either the past or future execution of the program. Careful selection of preemption points can ensure that migration does not occur at an inconvenient time. As an example, the **memcpy** procedure is non-migratible, but its effects are confined within the procedure and do not cause migration problems as long as migration does not occur during its execution.

- Type-safety disallows the existence of pointers that refer to objects of a different type. With migratibility, we are permitted to have these pointers (in

particular, `void` pointers), as long as they are used for pointer storage only, and are never dereferenced.

- Pointers that refer to an illegal address are considered a problem for both type-safety and migratibility. However, if we assume that the source and destination machines have the same data type sizes, this is no longer a problem for pointers that refer to heap blocks. That is, if a pointer refers beyond the end of valid heap block (either into invalid memory, or another heap block), we can translate the pointer relative to the beginning of the entire heap, rather than a particular heap block. This solution does not achieve type-safety, as it is still possible to violate the type system by dereferencing the pointer, but it does achieve migratibility.

- Storage of small integers inside the character data type is non-type-safe, but since most machines use the standard ASCII character set (therefore, no translation is required), the integer will retain the correct value.

5.4.3 Migratibility Is Not Source-code Specific

After studying all the issues presented in this chapter, an important comparison can be made between type-safety and migratibility. With type-safety, the only factor that contributes to the safety of the program is the source code itself, whereas run-time code and static analysis can only inform the programmer of type violations. However, with migration, the source code is not the only factor, as run-time support, type information, complexity of the migration tool, and differences between the machines all play an important role in determining a program's migratibility.

As an example, at first glance it appears to be non-migratible to have a pointer that refers to an unallocated memory location. Typically this problem would occur when pointer arithmetic causes the pointer to move beyond the end of an allocated array. To achieve migratibility, it is possible for the run-time system to record where the pointer was originally defined. When the program is migrated,

the pointer value is restored to the same position relative to its original location. Migratibility is achieved, but at no point does the source code of the program need to be altered.

This difference between obtaining type-safety and migratibility can be traced back to the definitions of each property. With type-safety, it is important to *avoid* any type violations, whereas with migratibility, we wish to *permit* the program to perform any action, as long as we can understand the effects of that action.

5.5 Detecting Non-migratibility

One of the purposes of constructing the Tui system was to determine the feasibility of automatically identifying the use of non-migratible language features. Although Tui has made good progress in this area, it does not provide the ultimate solution to this problem. To fully understand the theoretical limitations of detecting non-migratibility, and to provide a goal for future research, the limitations of migratibility detection will now be discussed.

5.5.1 Static Detection

Detection of non-migratible features at compile-time involves modification of the language compiler, or some other tool that only analyzes the static source code of the program. For the purpose of this discussion, detection of non-type-safe features is the same as detection of non-migratible features, although as previously discussed, the particular set of features is different. We must also assume that the language being analyzed is non-trivial, and will therefore contain language features that are problematic.

There are two types of static detection worth considering. Firstly, the source code can be scanned for the use of language features that will potentially cause problems. Even if a language feature exists in part of the program that will never be executed, an error will be reported. In the second situation, only those features

that are guaranteed to be reached are reported. For this discussion, the more general second approach will be taken.

The following proof by contradiction demonstrates that detection of non-type-safe language features is not always possible at compile-time. That is, although a program may clearly contain a non-type-safe feature, it is not always possible to determine whether that feature will ever be executed. This proof is similar to that of the well known halting problem.

Assume that we can construct a function `AnalyzeForSafety` *which takes the text of any program as its parameter and returns true if the program is type-safe, and false otherwise. This function must be written in a type-safe manner.*

Next, construct a program which combines the code of `AnalyzeForSafety` *with a main function as shown:*

```
Program A:
  main() {
    if (AnalyzeForSafety(A)) {
      perform_invalid_action();
    }
    return;
  }

  int AnalyzeForSafety(FILE *program_text) {
    ....
  }
```

Now, if program A declares itself to be type-safe, it performs a non-type-safe action. On the other hand, if program A declares itself non-type-safe, it exits normally without doing anything non-type-safe. Since a paradox occurs, we can conclude that `AnalyzeForSafety` *must be a non-terminating computation and therefore that the general problem of statically determining type-safety is non-computable.*

For migratibility, a similar proof can be used, except we must now consider the two implementations involved in the migration. That is, for a non-migratible

feature to be detected, the algorithm must be aware of which non-migratible features exist in the migration system. For example, if the process is to be migrated between two machines that have different character sets, the detection algorithm must report language features that depend on the internal representation of characters.

The `AnalyzeForMigratibility` algorithm would be written as follows:

```
Program A:
  main() {
    if (AnalyzeForMigratibility(A, ImplA, ImplB)) {
      perform_invalid_action();
    }
    return;
  }

  int AnalyzeForMigratibility(FILE *program_text, Impl A, Impl B) {
    ....
  }
```

An important requirement of both of these algorithms (`AnalyzeForSafety` and `AnalyzeForMigratibility`) is that they are themselves written in a type-safe (or migratible) manner. Even if a non-type-safe (or non-migratible) version could be written, it could always be translated into a type-safe (or migratible) version since both versions are written in a Turing complete language.

5.5.2 Dynamic Detection

It is frequently possible to detect non-type-safe features at run-time, when they were not detectable at compile-time. For example, an array bound violation can be reported as soon as it occurs, but can not be predicted until the value of the index expression is known. As with static detection, this type of dynamic detection is similar for both type-safety and migratibility, although as we will see, the desire to report errors may be different.

For type-safety, the programmer wishes to be informed of any possibility that a logical error may occur. The run-time system will check the use of the type system, and immediately report violations. One consequence is that some programs containing type violations may not actually contain errors. That is, the program will still produce the same result, regardless of whether or not the safety violation occurs. In reality, type-safe systems take the conservative approach of reporting all possible errors.

For migratibility, exactly the same situation arises. If run-time checking is too conservative, we may reject programs that are in fact migratible. On the other hand, if we do not reject programs at the point at which a non-migratible feature is used, we risk the possibility of incorrectly migrating a program. In either case, there is no simple approach to immediately determining whether or not to report an error. At best, it may be possible to delay the decision until a later point in time, when (or if) the non-migratible action actually disrupts the semantics of the program. No solutions to this problem have yet been proposed.

To explain why this issue is important, the `memcpy` procedure provides a common example of ANSI-C code. This procedure will take an array of any type, and copy it as if it were simply an array of characters. During the execution of this procedure, many non-migratible actions occur, although upon completion, the program will be in to a state where it is again migratible.

Being able to automatically detect these transient situations and distinguish them from non-migratible features that have a lasting effect would clearly help with migration of existing software. Tui has not addressed this problem, so future research would be beneficial for improving migratibility, and also has the potential for identifying localized type-safety problems.

5.6 Summary

The non-migratible features of ANSI-C have been analyzed to determine the factors that contribute to the migratibility of a program. When designing a migration system, each of these seven factors should be considered in order to determine the most appropriate method of resolving the non-migratible features of a language. The exact solutions depend on the requirements of the migration system's user.

An important contribution of this thesis has been to show that the concepts of type-safety and migratibility are not the same. Although the set of programs they describe are largely overlapping, there are exceptional cases that fall into one set or the other, but not both.

A program can be type-safe, but not migratible, if it depends on the characteristics of the underlying architecture. Even though a program may produce the same output on any of the available machines, the internal structure of data may differ, causing inconsistencies when the data is migrated.

A program can be migratible, but not type-safe, as long as the migration tool can guarantee that the program will perform in exactly the same way after being migrated. For example, some part of the migration system may understand the effects of a non-type-safe action, and will provide exactly the same effect on the destination.

Finally, it is shown that unlike type-safety, migratibility is not simply a property of the source language, but is dependent on the seven design factors: Language Restriction, Static Analysis, Run-time Code Checks, Run-time Code Support, Amount of Type Information, Functionality of the Migration Tool and Machine Similarities.

Chapter 6

Target Code Optimizations

One of the important assumptions made by Tui is that when a process is paused for migration (at a preemption point), it is possible to fetch the active state, and correctly restore that state on the destination machine. To be precise, when a program is compiled for multiple architectures, each version must contain the same set of preemption points. For each of these points, the set of variables and temporaries must be identical. Failure to do this will cause Tui to incorrectly retrieve or restore values.

If compiler optimizations are to be used, this assumption cannot always be made. In order to reduce code size or increase speed, optimizations will routinely rearrange or remove program code, remove unnecessary storage, or potentially add new temporary variables to avoid recalculating expressions. These code modifications can violate the assumptions made by Tui.

Although optimizations are not a primary focus of this thesis, their existence is considered important enough to justify a survey and brief analysis of their use. This chapter provides a discussion of the issues that arise when migrating optimized programs. It is necessary to study the effects that optimizations have on the program code, then if necessary, determine how to counteract these problems. In many cases, it may be straightforward enough to move, delete or create type information so that

it accurately reflects the effect of the optimization.

Firstly, a review of problems encountered in the related field of optimized code *debugging* will be given, and their relationship to *migration* of optimized code will be described. Next, the optimizations that are provided by the Amsterdam Compiler Kit will be described, and solutions will be presented. Finally, the problems introduced by more recent optimization techniques will be discussed.

6.1 Why Optimizations Cause Problems

Attempting to migrate code that has been automatically optimized by the compiler is similar to debugging code that has been optimized. In both cases, it is necessary to ascertain the current execution location, and be able to accurately retrieve the required data values. Optimized code debugging has been addressed [32] [19] [7] [34], resulting in several fairly good solutions, although none have proven to be totally reliable.

The general problem encountered with debugging is that optimizations will change the program structure, or remove parts of code such that the final program is noticeably different. Even though the eventual result of the program must be equivalent to that of the unoptimized program, the fine granularity at which programmers use a debugger to examine code will frequently allow them to observe the changes in structure.

Some of the potential problems that have been identified are:

- Even though a programmer can set a break point at a certain place in the source code, the corresponding object code may have been removed due to dead code elimination, or have been moved due to code motion. The program will not stop at the location the programmer had expected.
- Assignments to variables may be removed, or moved to a different location.

When a programmer wishes to display a variable's value, the *expected* value at the current location may not have actually been assigned yet, may have been reassigned with another value, or may not ever be assigned.

- Due to live variable analysis, a variable may share storage with other variables, as long as their lifetimes are mutually exclusive. Even if a variable is no longer current, the programmer will still wish to view the correct value, and would be misled if the debugger supplied the value of the wrong variable.

- If a bug has been located in a currently active program, the programmer might choose to assign a different value to a variable (from within the debugger) so that future expressions will evaluate correctly. If common subexpression elimination has been used to calculate an expression's value in advance (and is stored in a temporary variable), changing a variable will not have the desired effect.

With these problems in mind, the focus of optimized code debugging has been to provide an accurate method of mapping the program's current state back to the view of the program as perceived by the user. Essentially it becomes necessary to reverse the effects of all the optimizations at run-time, so it appears that the program has not been optimized.

The current solutions to this problem are based on identifying whether data values are *current* or *non-current.* If a variable is current, retrieving the correct value is trivial. If a variable is non-current (that is, its use has been optimized), it may not be possible to retrieve the expected value, but various techniques can be used to attempt to identify what the programmer expects to see. In the worst case, the programmer will be given potential values, along with a warning that the value may be incorrect due to optimization.

Migration of optimized code is similar, but there exist two differences that are significant enough to mention. Firstly, the uncertainty surrounding non-current

variables is not acceptable for migration. Although it may be sufficient for a programmer who is debugging a program to be told of a variable's possible value (rather than a definite value), the migration system could not function reliably under this condition. A more precise method of determining the correct value is needed.

Secondly, a migration system does not require interaction with a user, so there is no need to reestablish the user's view of the program. Instead, there must exist a way of mapping the state of the program (temporaries as well as variables) between the various versions of the software. Any number of optimizations may occur, as long as this property holds.

In the next section, we will examine the optimizations supported by the ACK compiler. The majority of these optimizations are performed at the intermediate code level, before any final code is generated. Because of this, the optimizations will not present problems for the migration tool, assuming that all the necessary type information (preemption points, call points, and variable descriptions) are added, deleted or modified as necessary.

6.2 Optimizations Supported By ACK

The Amsterdam Compiler Kit provides three separate optimization tools, although only one of them is used by default. The two remaining tools provide a variety of a optimizations that can be selected as desired by the programmer.

The optimization tools are as follows:

1. `em_opt` — This *peephole* optimizer acts at the intermediate code level, before any machine specific code is generated. It does not change the overall structure of the program, but instead simplifies instruction sequences within basic blocks (that is, between consecutive branch instructions or labels). The most noticeable optimization performed by this tool is constant folding.

2. em_ego — This *global* optimizer for intermediate code performs analysis of an entire procedure and therefore can rearrange the order in which basic blocks appear, or determine whether blocks should be removed entirely. There are nine separate optimization algorithms supplied by this tool, as well as several general purpose utilities for supplying control flow and data flow analysis information.

3. top — This tool is similar in style to em_opt, but operates at the machine specific level, rather than on intermediate code. Within each basic block of the final code, instructions may be deleted or replaced in order to improve the performance. For each processor type, top will perform a different set of operations, depending on the areas of efficiency of that particular architecture. However, ACK's optimizations are fairly simple in comparison to those possible with modern processors (such as support for pipelining or branch prediction).

Neither em_opt or top have any effect on the migratibility of object code. They concentrate entirely on optimizing sequences of code that appear within basic blocks. They are not capable of removing entire blocks of code, or procedure-wide temporary variables. The notable exception is that temporary values used during evaluation of expressions are often removed, but since the lifetime of these values is within the bounds of a basic block, there is no need to maintain their information. To help confirm this, em_opt has been used during the entire development of Tui, and has not caused any negative effects.

The global optimizations supported by em_ego do require changes to the type information generated by the compiler, but since they only operate on intermediate code, all versions of the compiled software (across the different architectures) will have an identical structure. Given this fact, ensuring that the type information is updated to reflect the outcome of the optimizations is enough to guarantee successful migration.

Most of the optimizations supported by em_ego require that type information

either needs to be moved around the program, removed from the program, freshly created, or modified. In only one case does an optimization not affect the type information in any way. The following list describes the optimizations and how they affect the type information.

1. Optimizations that require the removal of type information.

    - *Use definition analysis* – Performs constant and copy propagation. For example, if a conditional statement such as `if` or `while` always has a *false* value for the condition, the enclosed block of code can be removed. This optimization requires that all type information associated with that block also be removed.
    - *Live variable analysis* – Removes assignments to variables when the new value will never be used. In most cases, this will result in the removal of a single assignment statement, but in the extreme case where this assignment was the only reference to the variable, all type information associated with that variable should be removed.

2. Optimizations that require the creation of type information.

    - *Common subexpression elimination* – Calculates an expression once, and stores the value in a temporary variable to avoid costly recalculation at a later point in the program. Since this requires the introduction of a temporary variable, type information must be created to describe this new storage location. In compilers where the optimization phase is separate from the type-analysis phase (as is the case with ACK), obtaining this type information can be extremely difficult.
    - *Strength reduction* – Changes expensive operations (normally within a loop) into cheaper operations. Often this will result in the creation of temporary variables to avoid recalculating an expression in every loop

iteration. For example, when traversing an array, an indexing calculation could be replaced by less costly pointer arithmetic. As with common subexpression elimination, new temporary variables require the creation of type information.

3. Optimizations that require the movement of type information.

   - *Inline substitution* – Instead of actually making a procedure call, the body of a small procedure is inserted into the program at the point it is being called. This avoids unnecessary overhead associated with creating a new stack frame. Any type information within the original procedure must be relocated into the calling procedure. Some information, such as the frame pointer offset for local variables, will need to be updated.
   - *Branch optimization* – Rearrange the structure of the code generated for a looping construct (such as `while` or `do`) so as to minimize the number of branch instructions that need to be executed in each iteration. This optimization will change the ordering of the basic blocks, so all type information will need to be moved to reflect these changes.
   - *Cross jumping* – Factor out common blocks of code that appear within `if` statements. That is, if both blocks of code (for the true and false cases) end with an identical sequence of instructions, it is possible to move that code outside the `if` statement, hence reducing the size of the code. This will require that one copy of the code (and the type information) be moved outside the `if` statement, and that the second copy be deleted.

4. Optimizations that require modification of type information.

   - *Register allocation* – This optimization uses knowledge of variable usage to make suggestions as to which of them would benefit from being stored in registers. Even though no changes are required at this point in the

program's compilation, it is the final code generator's responsibility to place variables into registers, and to update the type information to reflect each variable's chosen location.

It is worth noting that the common *stabs* system is not sufficient to fully describe the structure of a program, particularly when register usage has been optimized. A modern description system, such as DWARF [4], is able to more accurately describe how values are stored.

5. Optimizations that require no changes.

   - *Stack pollution* – When a procedure call is made, it is the calling procedure's responsibility to remove the arguments from the stack. Stack pollution will avoid this time consuming step, at the cost of wasting stack space. For migration purposes, no type information needs to be changed.

6.3 Other Optimizations

Although the optimizations supported by ACK provide fairly good coverage of the traditional optimization techniques, there still exist a variety of algorithms that have not been considered. This section will discuss the issues involved in performing machine dependent optimization and describe some proposed solutions for obtaining migratibility in the presence of optimization.

As previously mentioned, it is acceptable to apply any number of machine independent optimizations to a program, as long as each version of the software is optimized in the same way. However, to achieve high performance, a program's instructions must be tailored to suit the underlying architecture. Therefore, each version of the program would be optimized differently and it may no longer be simple to map one architecture's version of the software to another. A recent survey [9] contains a comprehensive list of optimizations, for both sequential and parallel

architectures. Both machine dependent and machine independent optimizations are described.

A common issue is that of instruction scheduling. Most modern processors allow for some degree of parallelism, as they support the concepts of instruction pipelining, branch prediction, or super-scalar execution. In these cases, instruction sequences must be chosen to maximize the use of the processor's features. As a consequence, different optimizations are required for different processors. Algorithms for performing optimal selection of instructions are surveyed in [40].

An important consideration about machine dependent instruction scheduling (for pipelining and super-scalar execution) is that it often occurs within basic blocks, implying that it occurs between successive preemption points. If a guarantee can be made that the semantics of the code between successive preemption points is consistent across all architectures, any number of these optimizations can be performed.

At a higher level, a program's performance can be affected by the data access patterns expected by the processor. For example, a matrix multiplication algorithm will benefit from knowing how the processor's cache and external memory operates. Any loops in the program could be configured so that the optimal access pattern is used, or the data itself could be stored with an alignment padding that will avoid memory conflicts. The optimized code, such as the loop bounds test and associated variables, is specific for that architecture and can not easily be translated to another machine that has different requirements.

When these larger scale optimizations are performed, removing specific preemption points will ensure consistent semantics. For example, a fragment of code that performs matrix multiplication will contain several loops, although the exact specification of each loop will depend on the target architecture. To ensure successful migration, all preemption points within the code fragment should be removed. Only points that appear before or after the optimized code will allow a straightforward

mapping of state.

There are two significant problems that arise if preemption points are discarded to allow for optimizations. Firstly, it is now necessary to cross reference all machine dependent optimizations with all architectures. If an optimization on one machine requires that a preemption point be removed, all other architectures must be instructed to ignore that preemption point. Secondly, removing preemption points may introduce a substantial delay in halting the program for migration, or the program may terminate before a preemption point is reached.

A more ambitious attempt at maintaining consistency at preemption points is in the creation of *bridging code* [62]. When a machine dependent optimization causes the state of one version of the software to differ from the remaining versions, a small amount of code can executed to restore consistency. For example, the destination machine may have used common subexpression elimination to avoid recalculating an expression, whereas on the source machine, recalculating was considered more efficient than creating a new temporary variable. When migrating the process, the bridging code will need to derive a value for the temporary variable that only exists on the destination machine.

For the *bridging code* concept to be implemented, there must exist a method for comparing the state of two programs at each of the preemption points. If there are any differences, suitable code must be generated to map one state to the other. To avoid creating bridging code for every possible combination of architectures, it would be necessary to define a common state and only generate code to bridge *to* and *from* that state.

6.4 Summary

By studying the effects of optimized code generated by ACK, and by studying the literature, several observations have been made about the effects of code optimization on the migratibility of a program. Any number of machine independent optimiza-

tions may be performed, as long as the code for all architectures is optimized in the same way, and all type information is updated to reflect the effect of the optimization. Machine dependent optimizations are not as easy to deal with, but the proposed solutions of preemption point selection and generation of bridging code are worth further investigation.

Chapter 7

Migration of Other Languages

Throughout this research, the ANSI-C language was used when constructing migratible programs. This choice was made due to the popularity of the language within the UNIX environment, the availability of a suitable compiler, and most importantly because ANSI-C is often considered to be one of the least type-safe languages in common use, therefore uncovering the largest number of non-migratible language features.

Studying other languages is important for several reasons. Not only may each language have its own peculiar features that may hinder migration, but it may provide new programming constructs that supersede the non-migratible features identified in ANSI-C. For example, the FORTRAN language provides *assigned goto* statements that cause migratibility problems that are not apparent in ANSI-C. Additionally, C++ provides the *new* operator that resolves ANSI-C's problem with untyped malloc blocks. Future migration systems that wish to migrate programs written in these languages will benefit from the discussion provided in this chapter.

The languages that have been surveyed are:

- *FORTRAN* – Given that this was one of the earliest programming languages, and certainly one of the most popular, a large amount of FORTRAN software

exists. Typically this language is used for numerical programming.

- *Pascal* – Although not as popular as FORTRAN, Pascal was aimed at those who wished to maintain a good programming style. Even though it is not totally type-safe, many attempts were made to remove language features that could cause programming bugs.

- *C++* – This is becoming an extremely popular development environment, given that it supports the object-oriented paradigm, while at the same time remaining upwardly compatible with ANSI-C.

- *Ada* – A language designed by the U.S. Military for safety critical applications. Although it is still not a very common language (due to its history and the language complexity), public domain implementations may change this.

- *Scheme* – This is very different in style to the other languages in this survey, as it represents both interpreted and functional languages.

- *Java* – Due to its current popularity, its type-safety, and that programs are intended to be downloaded across a network, the migratibility of Java is worth analyzing.

For each of these languages, only a discussion of the features that distinguish the language from ANSI-C will be presented. For the sake of comparison, section 7.7 provides a summary (in table form) of whether or not each language contains each non-migratible feature identified in chapter 4.

Even though some language features are not supported by the current version of Tui, they will only be listed if they are truly non-migratible. For example, Ada has the concept of a *task* that permits concurrent programming. Although Tui assumes that only one task exists (the main process itself), dealing with multiple tasks would be straightforward, and is therefore migratible. On the other hand, FORTRAN frequently uses integers to store machine addresses, without providing

a method of distinguishing them from normal integers; this feature clearly precludes migration.

Finally, each of the languages was surveyed by examining a language description document, rather than by analyzing common programs (as was done with ANSI-C). In most cases, the official language document was used, but where this wasn't possible due to cost, a suitable alternative was found. Also, with the older languages, formal specifications were not common, so it is sometimes difficult to determine whether or not a particular feature is non-migratible.

7.1 FORTRAN

FORTRAN has grown over time, and several standard versions have been produced. This survey primarily focusses on FORTRAN-77, with a small amount of discussion on the more recent FORTRAN-90. Although these standards are formally described in documents produced by the ISO, they are only available at a cost. Instead, the language reference for a popular FORTRAN compiler [20] and an application programming textbook [53] were examined.

Given that FORTRAN was created before any significant research had been put into language design, many of the features can be considered primitive. For example, pointers and dynamic allocation do not exist for general data structures, and in some cases, integers are used to store machine addresses. However, as newer versions of FORTRAN were produced, many non-type-safe features were labelled as obsolete.

The specific features of FORTRAN that hinder or aid migratibility are:

- *Common blocks* – This feature is a primitive method of sharing data between different subprograms. Within each subprogram (a subroutine or function), the programmer can specify a set of variables that will appear in the common block, as well as their type and order within the memory layout. The same

block of memory is shared between all participating subprograms. Common blocks are non-migratible since within each subprogram, the programmer is permitted to specify a different set of variables and types. The migration tool will not know which set of types is correct.

- *Equivalence statements* – These are similar to the union data type in ANSI-C, but are not restricted to a well known area of memory (such as defined by the union itself). That is, a programmer may state that any variable should be stored in the same memory location as any other variable in the program. Normally this will produce non-migratible code, but in some situations, such as making a complex number equivalent to two floating point values, equivalence is acceptable.

- *Assign statements* – In order to branch to program labels, the **assign** statement permits a programmer to store a machine address within an integer. It is not legal to interpret this value as an integer, but there is no means of stopping a program, or a migration tool, from doing so. The only legal use is by an instruction such as `go to` that interprets the integer as a pointer.

- *Alternate return addresses* – When a subprogram terminates, instead of always returning to exactly where it was called from, it is possible to supply alternate return addresses. These addresses are passed to the subprogram as integers, rather than as a special pointer data type.

- *Passing procedures as variables* – A pointer to a subprogram can be passed as an argument to another subprogram. However, when this pointer is dereferenced, it is not possible to check whether the correct number or type of arguments are being supplied.

- *Separate compilation* – Although FORTRAN-77 has the same separate compilation problems as ANSI-C, FORTRAN-90 introduces a *module* feature that permits type checking across files.

- *Allocatable arrays* – FORTRAN-90 introduces dynamic allocation of arrays, and since it is possible to determine the size of an array, migratibility can be achieved.

7.2 Pascal

Pascal [37] was created after considerable discussion about the Algol family of languages. The intention was to provide a clean language, with a moderate set of capabilities, but without any of the features that would typically cause programming errors. Type-safety was almost achieved, although a few small problems are still present.

Although Pascal and its derivative, Modula-2, are frequently used as teaching languages, very little industry support remains.

The interesting features of Pascal are:

- *Variant records* – Similar to ANSI-C unions, although a tag can be used to state which of the fields is currently active. In general, the use of this tag is optional, although for migration, the run-time system must ensure correct access to the data.

- *Pointers* – Pointers are available, but the set of operations is limited in comparison to ANSI-C. The significant differences are that pointers may not be converted to or from integers, and pointer arithmetic is not permitted. It is therefore not possible to generate a pointer to an arbitrary memory location.

- *Dynamic allocation* – All dynamic data structures are created via the `new` procedure, using a previously declared pointer variable to determine the type of the data. Assuming this type information is stored within the heap block, a migration tool can be guaranteed to retrieve the correct heap data.

- *Dangling pointers* – Dynamic memory is deallocated by the `dispose` procedure. When this is used, all pointers to that area of memory become undefined. This introduces the same problems as the `free` function in ANSI-C. Since it is not possible for a pointer to refer to a local variable, dangling pointers can not be generated when a function terminates.

- *Arrays* – It is always possible to determine the size and type of an array, although it is not specified as to whether run-time bounds checking is a required part of the language. If it isn't used, illegal array accesses could corrupt memory and create non-migratible values.

- *Separate compilation* – Pascal programs must be contained within a single source file, so type checking between files is not an issue.

7.3 C++

C++ [63] is almost a complete superset of ANSI-C, so most of the non-migratible features identified in section 4.2 are still relevant. However, C++ introduces new concepts such as object-oriented programming and exceptions, that allow for new migratibility issues. Fortunately, most of the new features in the language improve the migratibility of a program, rather than introducing new problems.

- *Mismatched parameters* – Unlike in ANSI-C, a function must now be declared before being used. Also, an empty parameter list in ANSI-C suggested that the procedure could accept any number of arguments, whereas in C++ this type of procedure may not accept any parameters.

- *Dynamic allocation* – The `malloc` procedure of ANSI-C has been replaced by the `new` operator. The programmer must specify the type of a dynamically created object, and in the case of an array, the number of elements. As in Pascal, having this extra type information removes the problems associated with

untyped dynamic allocation. One minor complication is that the programmer is still permitted to use their own allocation method, or to redefined the `new` operator, although this practice is not common.

- *Exception handling* – The `setjmp` and `longjmp` procedures of ANSI-C have been replaced by a well defined exception processing mechanism. The same information (that is, a program counter and a stack pointer) must be saved, but since the mechanism is now an integrated part of the language, there is no confusion over the use of any storage, as was seen with the `jmp_buf` structure.

7.4 Ada

Ada was designed with the concerns of program reliability and maintenance. It was originated by the U.S. Military as a standard language for all software development, although it has now reached commercial environments. The most recent standardization of the language [1] was performed in 1995.

The language supports concepts such as separate compilation using *packages* and concurrency and synchronization using *tasks*, as well as the more common features of object oriented programming and exceptions. Due to this extensive range of features, the syntax and semantics are far more complex than those of most languages.

Although a significant effort has been made to achieve type-safety for most programs, Ada still provides mechanisms for accessing the low-level features of the underlying architecture. For example, it is possible to determine the size and alignment of data objects, as well as to enter machine code instructions into the program. Even though these features could potentially introduce the same non-migratibility problems of ANSI-C, they are much less common, and are not an integral part of the language.

The following specific language features are important for migration, and are signif-

icantly different from concepts of the other languages surveyed so far:

- *Dynamic Allocation* – As with C++ and Pascal, dynamic allocation relies upon the `new` operator, so the type of all heap data is well known. To ensure that dangling pointers are not possible, a series of rules are checked at either compile-time or run-time. Garbage collection is suggested as an implementation dependent feature, but if the user wishes to explicitly deallocate memory they may do so using an *unchecked* deallocator that may introduce dangling pointers.

- *Run-time checks* – Run-time bound checks are performed on array element accesses and on field accesses for discriminated record types (similar to Pascal's tagged variant records). If any violations occur, an exception will be generated.

- *Data type details* – A program is able to determine the size and alignment of any data object. In some cases, it is able to select the desired size of data values. For example, the accuracy of a real number can be requested by specifying the number of digits to be stored. Migration of this type of program will not be possible if the destination machine is not able to support the accuracy required by the program. In this case, the destination machine would not have been able to execute the program correctly, even before migration is considered.

- *Type-safety violations* – Features exist for performing unchecked type conversion and unchecked creation of access type values (that is, pointers). These features are similar to those of ANSI-C, but would only be used in exceptional circumstances, rather than for general use.

7.5 Scheme

Scheme [17] provides a very different programming environment from the other languages discussed in this survey. Typically a Scheme program is entered interactively, rather than being compiled into machine code in advance. If an expression is entered by the user, the result is calculated immediately, whereas procedure and data definitions are stored in internal data structures for later use. In this sense, Scheme is representative of the set of interpreted languages.

Scheme is based on the theory of lambda calculus. The language provides very few features, but those that exist are very general in nature. Because of this, programs can easily be written using programming paradigms such as functional, procedural or object oriented styles.

A dynamic type system is used, as opposed to the static typing of most other languages. Instead of declaring each variable to have a well defined type, a variable may hold a value of any type. Each value in the program's memory must be tagged with a type descriptor, so that a value's type can be verified before any operations are performed. Because of this, Scheme is totally type-safe, and no source level non-migratible features exist.

The following language features ensure that type-safety is obtained:

- *Pointers* – Even though pointers are used extensively in the implementation of Scheme, and can be seen within some of the language's constructs, there is no explicit means of creating a pointer value. This restriction solves several of the problems witnessed in ANSI-C.

- *Dangling references* – Although dynamic allocation is performed by operators such as `cons`, there is no method for explicitly deallocating memory. A garbage collection algorithm will locate and deallocate all heap blocks that are no longer accessible. In this environment, a pointer can never refer to a

deallocated heap block.

- *Range checking* – Operations on the list, vector and string data types will perform appropriate run-time range checks to ensure that data corruption is not possible.

- *Format conversion losses* – Numerical values in Scheme are not limited by the word size of the underlying processor, as they do not use the processor's standard storage format. The exact extent of a number's range is dependent on the particular Scheme implementation, and is often confined only by the machine's memory size. For migration purposes, we can be certain that both source and destination machines will use the same storage format.

- *Continuations* – Scheme uses the concept of *continuations* to provide the effect of `setjmp` and `longjmp`. Since continuations are built into the language, migratibility can always be achieved.

Although Scheme is migratible at the source level, problems may arise in the implementation of the language. An inefficient implementation may choose to translate the Scheme program into basic lambda calculus, and then provide a low-level interpreter for executing the calculus-style instructions. Assuming that the interpreter and its data structures are written in a migratible fashion, migration of a Scheme program is always possible.

A second method is to translate the Scheme program into an equivalent program in C (or another suitable language). This removes the interactive feature of the language, but provides efficient program execution on any architecture that supports a C compiler. Migratibility of Scheme now depends on the migratibility of the C code. This technique was demonstrated in section 4.1 to migrate programs written in the ST language.

A final method involves incrementally compiling the Scheme code into machine code, as procedure definitions are entered. Although both speed and interac-

tivity are still available, the run-time system is very machine specific. Migration of a Scheme program relies on the ability to recompile the procedures for use on the target machine, and to identify suitable preemption points.

7.6 Java

Because of the popularity of the internet, Java [30] has quickly achieved widespread use, as it allows programs to be downloaded from a remote site and automatically executed on the local machine. The source language is compiled into virtual machine instructions [42], rather than depending on a real hardware architecture. A virtual environment has been designed to provide simple access to internet resources, as well as to present a machine independent view of the graphical user interface.

The Java language is similar in design to C++, in that it uses similar syntax, and also supports object-oriented programming. Only language features that were considered to be well-tested and useful were included in the language definition. Also, since Java programs are intended for execution on remote machines (that is, anywhere in the internet), type-safety is considered extremely important. Without type-safety, there exists a danger that a malicious Java program could disrupt a host machine that has innocently downloaded the program.

Given that Java also has roots in the Scheme language, a similar type of dynamic allocation system is used. Object references are created by the `new` operation, and a garbage collection mechanism is called upon to remove outdated heap blocks. General manipulation of pointer values is not permitted. In addition, pointer variables (known as *object references*) are implicitly initialized to the null value.

An unusual feature of Java is that all data type formats and sizes are clearly specified. In other languages, these decisions are normally left to individual language implementations so they may obtain optimal performance on their specific architecture. When downloading a precompiled Java program, a common format guarantees that the bytecodes will be interpreted in the same way on all architectures. For a

similar reason, this requirement is also important to achieve migratibility.

The implementation considerations for Java are similar to those for Scheme. In most cases, Java bytecodes will be downloaded to the local host, then interpreted by a web browser. For better performance, bytecodes could be translated into another language, or be compiled to native machine code. Even though interpretation is currently the most common method, and retrieving a program's state is therefore trivial, Java does not yet support process migration.

7.7 Summary of Details

The previous sections have presented a brief overview of each of the languages, with a discussion of any problems that were significantly different from those identified for ANSI-C. This section provides a summary of these differences, by listing the features given in section 4.2 and stating whether they are present in the other languages.

Without giving a detailed analysis of each non-migratible feature, it is difficult to say whether particular problems exists or not. For example, although C++ programmers may still use the `malloc` function, it has been superseded by the `new` operation. Also, FORTRAN does not directly support the union data type, but the equivalence statement can be used to provide the same functionality. This summary should be used to determine the quantity and general type of non-migratible features that exist in each language, rather than as a detailed analysis.

Note that two of the problems listed in section 4.2, that is, *1.4 Reused variable storage* and *2.3 Untyped Internal Constants*, have not been included in this summary. Without having an implementation of the language to study, it is not possible to determine whether these problems will occur. They are certainly not discussed in the language documentation.

Language Feature	ANSI-C	Fortran	Pascal	C++	Ada	Scheme	Java
1.1 Union data type	√	√	√	√	X	X	X
1.2 Pointer casting	√	X	X	√	X	X	X
1.3 Mismatched parameters	√	√	X	X	X	X	X
1.5 Separate compilation	√	√	X	√	X	X	X
2.1 Generic (void) pointers	√	X	X	√	X	X	X
2.2 Untyped Dynamic Allocation	√	X	X	X	X	X	X
2.4 Variable Argument Lists	√	X	X	√	X	X	X
2.5 argv array	√	X	X	√	X	X	X
2.6 Setjmp and Longjmp	√	X	X	X	X	X	X
3.1 Casting integers to pointers	√	√	X	√	X	X	X
3.2 Pointers to Illegal Memory	√	X	X	√	X	X	X
3.3 Dangling pointers	√	X	√	√	X	X	X
3.4 Uninitialized pointers	√	X	√	√	X	X	X
3.5 Omitted return values	√	√	X	√	X	X	X
4.1 The "sizeof" Operator	√	X	X	√	X	X	X
4.2 Double Meaning of "char"	√	X	X	√	X	X	X
4.3 Format Conversion Losses	√	√	√	√	X	X	X

7.8 Language Design for Migratibility

To complete this survey of migratibility in common languages, a brief discussion of how language design affects migratibility is important. Even though the majority of existing software is written in FORTRAN and C, the other languages are in common use, and some will continue to become more popular in the future.

The most notable conclusion, and perhaps the most obvious, is that the number of migratibility problems is related to the number of type-safety problems. In the design of FORTRAN and C, very little effort was made to ensure that data types are not used in an incorrect way, such as using integers variables to store pointer values. These programming habits, and the inability to detect them, has

lead to a generally accepted programming style that relies upon the use of non-type-safe features.

In more recent times, not only has type-safety become an important design feature, but programming style has improved. Languages such as Pascal, Ada, Scheme and Java, limit the programmers ability to perform non-type-safe activities, and therefore require a programming style that has proven to be more suitable for migration. Although C++ and Ada provide the same low-level facilities of ANSI-C, these features tend not to be as commonly used.

Finally, language implementation is an important factor for migratibility. In languages, such as Java, where the storage format of data values is well defined, migration between different architectures will never fail due to data conversion losses. This issue has not typically been considered as part of type-safety, but may become more common as programming languages begin to support distributed computing.

Chapter 8

Performance

Any software product intended for frequent use must obtain an acceptable level of performance. This chapter demonstrates that heterogeneous process migration is capable of such performance, given the expectations presented by its users.

Firstly, a detailed analysis of the various components of Tui's `migrin` and `migrout` algorithms is given. Three test programs were written to stress various parts of the system. This analysis was used to derive the revised version of Tui by uncovering the performance weaknesses of the original algorithms.

Next, these same programs were used to determine the asymptotic complexity of the algorithms. The programs were executed with varying data set sizes, and then migrated. The performance of the algorithms is studied with respect to the size of the program being migrated.

Next, to demonstrate that migration is not limited to sample programs, two realistic applications (MicroEMACS and the matrix multiplication package) were migrated. This shows that long-running processes can be migrated in an amount of time that is relatively small in comparison to the total execution time of the program.

Finally, an example program is used to demonstrate that migration of a program over wide-area network (such as the internet) will provide much better

performance than the alternate option of remotely accessing the data.

8.1 Components of the `migrin` and `migrout` algorithms

To fully test the performance of the Tui algorithms, three different test programs (for Tui to migrate) have been created. Each was designed to test the complexity of the various components of Tui.

The three programs are:

- `fibonacci` – An inefficient recursive implementation of the Fibonacci algorithm that creates a large number of stack frames, each with a small number of local variables and temporaries. A single preemption point is placed so that migration will occur when n stack frames are active (n is the input parameter). This program tests `migrout`'s efficiency when scanning the stack.
- `tree` – Builds a binary tree of n nodes (n is a command line parameter). Numeric values are selected randomly and then inserted into the tree. Once construction of the tree has completed, migration will occur. This program tests Tui's ability to deal with a large number of heap blocks.
- `arrays` – 50 character arrays (of user specified size) are dynamically allocated on the heap and then filled with characters. This test demonstrates the efficiency of encoding and reconstructing large areas of memory.

To demonstrate that Tui can correctly function on the four supported architectures, each of the programs was migrated. Figures 8.1 to 8.4 show the time taken by the main components of both the `migrin` and `migrout` algorithms for the `tree` program. In these tests, the number of tree nodes varies from 1000 to 8000. Although only the `tree` program is presented here, `fibonacci` and `arrays` produced similar results.

The exact machines are:

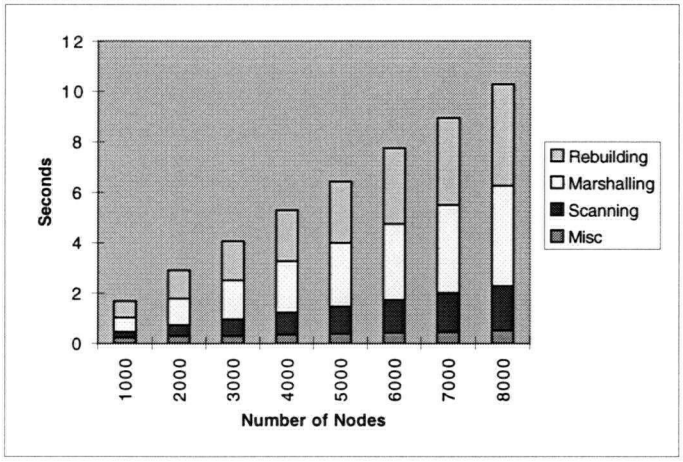

Figure 8.1: "tree" on Sun 4

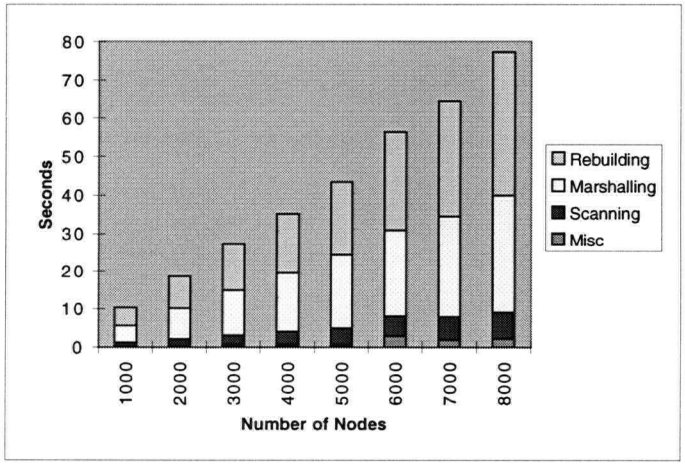

Figure 8.2: "tree" on Sun 3

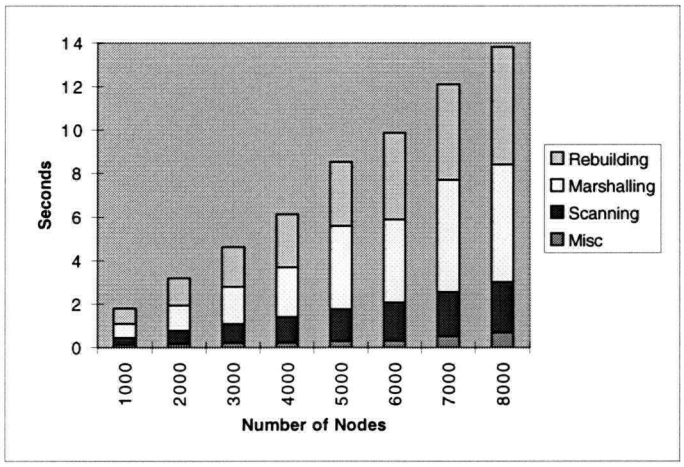

Figure 8.3: "tree" on i486

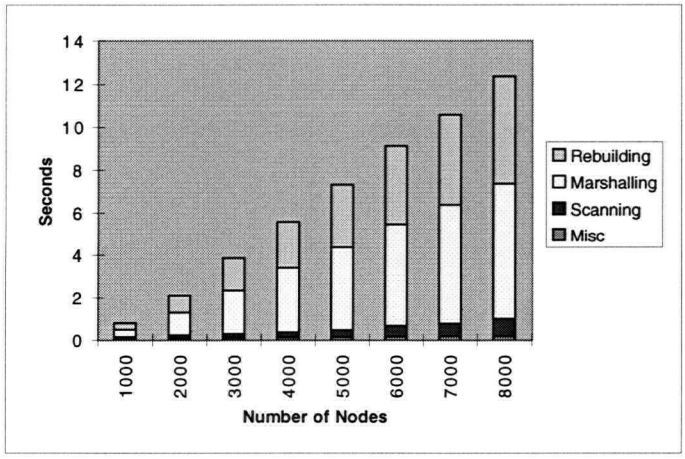

Figure 8.4: "tree" on PowerPC

- Sun 4/75 (SPARCStation 2)
- Sun 3/60
- i486 running at 50Mhz
- PowerPC 601 running at 66 Mhz

All measurements are averaged over 5 runs on an otherwise idle CPU. The machines have sufficient memory to avoid paging.

In this analysis, the total execution time is divided into the major components of both `migrout` and `migrin`. We must pay attention to the relative costs between the components and the growth of each component as the problem size increases. The following list gives an explanation of each cost.

- `Miscellaneous` – The time required to read the migrating program's memory image into Tui's address space, as well as the time to read the program's type information from disk. The memory image size will vary depending on the data size of the program, but the amount of type information will remain constant for any particular application.
- `Scanning` – The scanning of the memory segments and the construction of the value table. This cost depends on the number of individual data values that are located, not the size of those values.
- `Encoding` – The data values must be marshalled into the intermediate file. This cost depends on the total size of all data values, as well as the operating system's performance when writing to files.
- `Rebuilding` – This is the only component of the `migrin` algorithm that has been analyzed. Given the intermediate file, the new data and stack segments are constructed. The other components of `migrin`, such as reading the type information and reading/writing core memory is the same as for `migrout`.

Note that the final rebuilding of `migrin` can almost entirely occur in parallel with the scanning and encoding of the `migrout` phase. Therefore the total migration time will be less than the total time required over all components.

It can be seen that scanning, encoding and rebuilding are the major components of the migration cost. The miscellaneous costs of reading type information and the memory image is small enough to ignore. We next study how the cost of the three major components increases as the input size becomes extremely large.

8.2 Asymptotic Growth in Migration time

To examine Tui's performance when migrating realistically sized programs, each of the three tests was configured so that it would create a large memory image. Figures 8.5 to 8.7 show the contribution of the major costs (scanning, encoding and rebuilding) for various input sizes. The following list gives an explanation of the performance for each of the three programs. To avoid memory paging problems, all tests were performed on the same large machine (the PowerPC).

- `tree` – This program has close to linear performance for all three components. The makes sense for scanning and rebuilding, but for encoding we expect an asymptotic O($nlogn$) complexity due to the binary search that is done on the value table for every pointer. In these results, this extra complexity does not appear to be significant.

- `fibonacci` – The complexity is roughly the same as for `tree`, but the overall running time is lower.

- `arrays` – Since there are only 50 arrays, the scanning component requires an insignificant amount of time to locate them. However, since each array can be large (up to 50000 characters in our case), the encoding and rebuilding components are significant, although they will always have linear complexity.

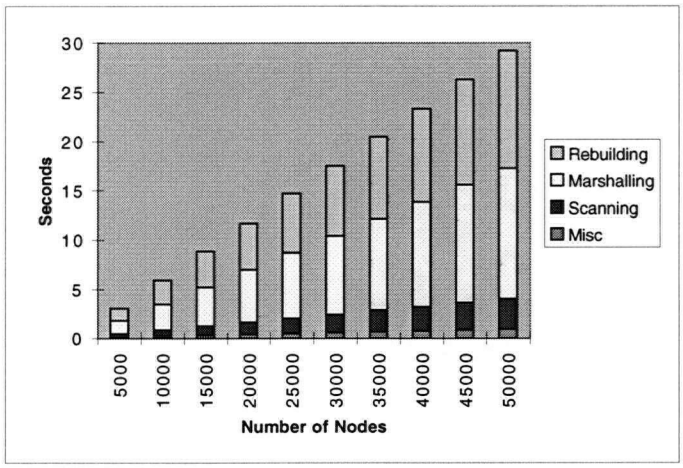

Figure 8.5: Growth of "tree"

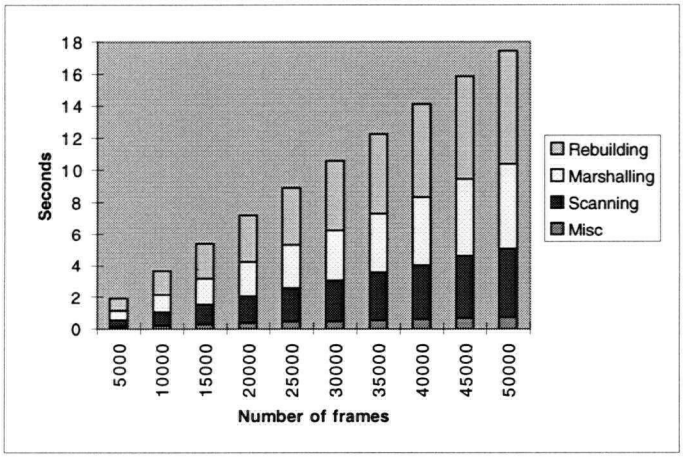

Figure 8.6: Growth of "fibonacci"

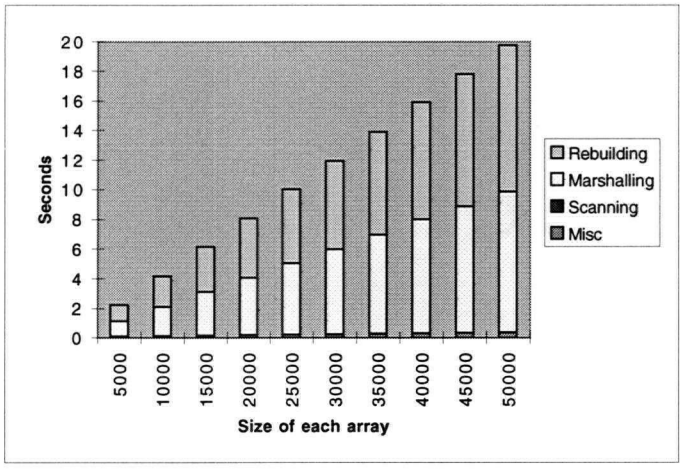

Figure 8.7: Growth of "arrays"

In the original version of Tui, the scanning component of the algorithm did not scale well for the fibonacci example. Due to the access patterns of adding data to the value table (implemented as a splay tree), the worst case performance of $O(n^2)$ was being observed. This problem motivated the revised version of Tui that uses an expandable table, rather than a splay tree, as well as requiring that data be traversed in a linear order.

The overall performance of Tui as seen by empirical results is approximately $O(n)$. Upon closer analysis, the most complex part of the algorithm requires $O(n + mlogn)$ time, where n is the number of data objects in the program, and m is the number of pointers. That is, each data item in the program must be examined once (to construct the value table), then for each pointer in the program, a binary search is performed to determine which object the pointer is referring to. The final stage, encoding the data values, can fairly time consuming, but will always have $O(n)$ complexity.

Tui has not reached the theoretical minimum complexity for a heterogeneous migration algorithm. Firstly, we know that it is not possible for an algorithm to

have complexity of less than $O(n)$, since we must examine and translate each of the n data items at least once. Secondly, if extra memory was available for tagging data items with their indicies, the process of determining which data item a pointer refers to could be done in constant time, rather than requiring a binary search. The lower bound on complexity for heterogeneous migration would therefore be $O(n)$.

8.3 Migrating Realistic Programs

The three test programs discussed so far were designed to determine the performance of the major components of Tui. However, to demonstrate that Tui is not limited to a small set of contrived programs, two applications that were not designed for migration have been used. The matrix multiplication package and MicroEMACS editor, that are described in section 4.1, demonstrate how easily "real" programs can be migrated.

In both these cases, it is important to consider the total expected running time of the program. A complex mathematical calculation involving matrices, may execute for many hours. A text editor may not consume much processor time, but it will often remain active anywhere from several minutes to several weeks. Migrating a program at a cost of up to one minute becomes beneficial when the matrix multiplication package moves to a faster machine, or the text editor moves to a location that is more convenient for the user.

Matrix Multiplication

In figure 8.8, the migration time is shown in relation to the dimension of each matrix. Since the matrices are two dimensional, a doubling in dimension will result in four times the memory usage. In the largest test shown, the memory usage of the program was approximately 4 megabytes, requiring a total of 20 seconds to migrate. The asymptotic growth of Tui's execution time remains linear, as expected.

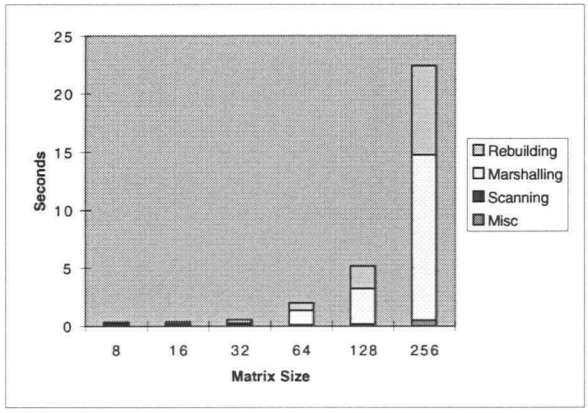

Figure 8.8: Growth of "matrix"

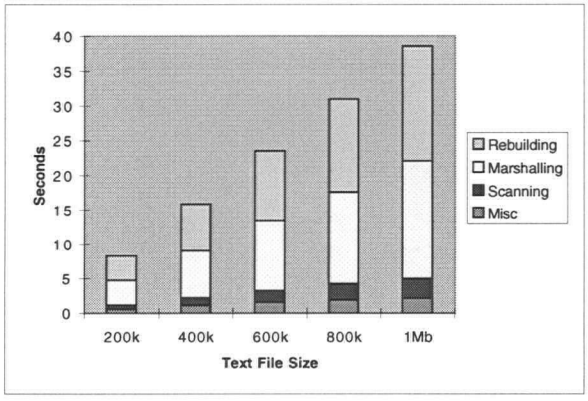

Figure 8.9: Growth of "uemacs"

The MicroEMACS Editor

In figure 8.9, the migration time is shown with respect to the size of the data file being edited. A file is stored as a linked list of lines, and each line is stored as an array of characters. The majority of the migration time is consumed by the marshalling and restoration of the data, rather than in scanning memory.

The complexity of the algorithm remains approximately linear, and the absolute execution time is reasonable when the total lifetime of the editor is considered. If the `migrout` and `migrin` phases were to be executed in parallel, a migration time of 20-25 seconds for a 1Mb text file could be achieved.

8.4 Remote Access Versus Migration

One of the primary motivations for using process migration is that communication costs can be reduced by moving a process closer to the data it is accessing. This is particularly true in a mobile or wide-area computing environment. To demonstrate the effectiveness, an experiment has been performed to study the cost of migrating the process closer to the data, in comparison to accessing the data over a wide-area network.

8.4.1 The Experiment

A simple version of the Unix `ls` command has been written and compiled for migration by Tui. Using the recursive listing option, the entire directory structure will be displayed. This program requires repeated access to the file server in order to fetch file or directory names. In this experiment, the directory being listed is `/usr/share/lib` on a Solaris system, containing 2136 files within 61 subdirectories.

Performing this experiment across the Internet would have been desirable, but due to the lack of a suitable remote site, this was not possible. Instead, the experiment was performed locally, with a Linux machine acting as the remote host,

a PowerPC as a local host, and a Solaris machine as the file server. The file server was modified to enforce a 240 millisecond delay for every request that came from the Linux machine. This is approximately the round-trip delay of sending network data from North America to Europe.

Before delays were introduced, the following execution time were recorded for both the Linux and PowerPC computers:

	CPU time (sec)	Real time (sec)
Linux	10	36
PowerPC	6	38

It can seen from these values that the majority of execution time is consumed by the inefficient Tui file server, rather than by the client program. The file server is accessed once to retrieve each file's name, and then a second time to determine whether it is a directory that should be recursively traversed.

When the artificial delay is introduced for the Linux machine, the following result is obtained:

	CPU time (sec)	Real time (sec)
Linux	10	1219

In this situation, the ideal scenario would be for the remote host's operating system to observe that the program is repeatedly spending 99% of its time waiting for data to arrive from a single remote file server. After a short period of time (such as 10 seconds), the operating system would initiate migration to a host that is closer to the file server (in this case, the PowerPC). The only concern is that migration time should not be significant when compared to the remaining time of the program's execution. In general, this is not possible to determine, but given that the program has already executed for 10 seconds, further execution may be likely.

The following results were obtained in the experiment:

Activity	Time (seconds)
Execute the program at the remote host until migration is initiated by the operating system	10
Perform the `migrout` operation on the Linux machine	4
Transmit the intermediate file (34949 bytes in size) from the Linux machine to the PowerPC. (Actual file transfers from Europe to North America were used to estimate this value)	10
Perform the `migrin` command on the PowerPC	2
Continue the program's execution on the PowerPC	36
Total time	62

It can be clearly seen that migrating the process, requiring a total of 62 seconds, is clearly better than leaving the process to continue remotely accessing the data, requiring 1219 seconds.

8.4.2 Discussion

In this example, it could be argued that because the process has moved away from its original site, the directory listing must travel back to the source at a high expense. In this case, the output is being produced sequentially, so there is no need for the program to delay while the data returns to the originating machine. However, if the output was being written to a non-sequential file, such as a database, this method would not necessarily reduce communication costs.

It must also be noted that obtaining a directory listing from a remote machine is best achieved by providing a specialized service on the file server. Rather than requiring the client to repeatedly access the file server, the service could complete

the entire listing and send the result to the client as a single stream of data. This approach is taken by a File Transfer Protocol server, as it only requires a single round-trip delay.

The disadvantage of providing this type of special service is that the expected functionality must be known in advance. On the other hand, allowing a client program to remotely access files and perform the service themselves, will ensure that any type of functionality is possible. By allowing migration of client programs, we can essentially install services onto the file server as needed.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

The Tui heterogeneous migration system has shown that migration is practical for software written in non-type-safe languages. The development of this system has provided insight into creating an efficient migration algorithm, as well as an understanding of what is required for a program to be migratible.

The original version of Tui made no assumptions about the source program, but it was not always possible to determine the type and size of dynamically created heap data. For the revised version, a restriction that was placed on the source program provided the means to solve this problem, and to create a more efficient migration algorithm. The revised version of Tui can migrate a program in $O(n + mlogn)$ time, where n is the number of data elements, and m is the number of pointers in the program.

A comprehensive list of the non-migratible features of the ANSI-C language have been identified by using Tui to migrate a set of application programs. This survey has not only provided an understanding of the exact problems that a migration system must solve, but it also demonstrates that realistic programs can be migrated.

A study of the concept of migratibility has shown that it is incorrect to assume a program must be type-safe before it may be migrated. Type-safety is an absolute property of the source language, whereas migratibility can also be achieved by modifying the run-time code and migration tool, or by making assumptions about the source and destination machines. These factors are an important consideration for the future design of heterogeneous migration systems.

Code optimization is an important issue affecting the migratibility of a program. A study of common optimization techniques has shown that although any number of machine independent optimizations may be applied to a program. Assuming that the compiler generated type information is correctly maintained by moving, creating or destroying information, there will be no affect on the program's migratibility. For machine dependent optimizations, migratibility may be achieved by either removing preemption points from the program, or by adding bridging code that will reverse the effects of the optimizations.

A survey of common programming languages has shown that older languages such as ANSI-C and FORTRAN contain a variety of non-migratible language features, whereas newer languages, that have type-safety as a goal, are much better suited for migration. The outstanding concern is that many languages do not require the same data representations across all architectures, as is useful for migration.

The performance of the Tui system has shown that the migration-time cost of heterogeneous process migration is noticeable, it can be considered negligible in comparison to the run-time costs of a program that we may wish to migrate. This is particularly true for programs, such as text editors or CPU intensive simulations, that execute for large periods of time. When programs are suffering expensive network costs (as in a mobile or wide-area environment), migration provides substantial savings over accessing data remotely.

9.2 The Effort Required to Create Tui

Constructing the Tui migration system was a time consuming task, although much of the effort was in determining the fine details of the migration algorithms. If a similar migration system were to be implemented, based on the design of Tui, the implementation effort would be greatly reduced. Such a system would need to be created to allow for the migration of programs written in significantly different languages, or for the use of alternate type information formats (as opposed to "stabs").

The following table shows the number of lines of ANSI-C code, as reported by the UNIX `wc` command, for each component of the Tui system. A complete description of these components will be given in Appendix A.

Program	Lines of Code
migrout	6757
migrin	5630
prdump	996
fileserv	1389
procserv	1620
migrate	737
show_points	832
tuiprep	6011
tuiprepprint	3073

In addition to this newly created software, a number of modifications were made to the standard Amsterdam Compiler Kit. The ANSI-C compiler frontend was augmented so that the extra type information (preemption points, call points and stack frame structure) was generated. For each target architecture, the relevant code generation, assembling and linking programs were correspondingly modified to allow for these new details.

A series of smaller modifications were made to the ANSI-C frontend, to warn or deal with the occurrence of non-migratible features. For example, when an incompatible pointer type cast is observed, a warning message will now be produced. Also, any call to the `malloc` procedure will now implicitly involve the generation and storage of type information.

Finally, the ANSI-C run-time libraries were modified for use by Tui. Some of the procedures, particularly those relating to operating system functions, were modified to make use of the Tui file server. In other cases, procedures were modified to remove the use of non-migratible language features.

9.3 Future work

Heterogeneous Process Migration is still a fairly new topic. One purpose of this thesis has been to identify the important issues in this field, and to suggest further research that could improve upon the existing techniques. Some proposed future work will now be presented.

9.3.1 A More Precise Migration Tool

The migration tool proposed in section 5.3 should be constructed and used in practical environments. This tool is based on the idea that maintaining and checking extensive amounts of type information will immediately identify non-migratible actions. It could provide accurate details of any non-migratibility problems that occur, in particular, the ability to display the section of source code that caused the problem. Although this tool would seriously impede run-time performance, it would only be used during development to verify the migratibility of a program. The final result would be a user friendly package that requires only minimal knowledge of migratibility issues.

9.3.2 Source Code Analysis and Modification Tools

Where possible, complex static analysis tools could be used to detect the use of non-migratible language features. Rather than simply reporting potential misuses (as is done by Tui), detailed analysis could provide a more accurate view of where migratibility problems are occurring. For example, when compiling the `memcpy` function, the tool should detect that the non-migratible features are only being used locally, and will not be a problem as long as preemption points are not placed within that function.

After analysis has been completed, modification tools could be used to automatically alter the source code. For example, since all calls to `malloc` must contain the `sizeof` operator, a tool could locate all occurrences of `malloc`, suggest possible types for the heap block, and point out any ways in which the heap block is not being used in a standard way. Some tools, such as the Unix `protoize` tool, can be used in their existing form.

9.3.3 Incorporate into an Operating System that Supports Migration

Tui currently functions as a user process within the Unix environment, which significantly limits its functionality. Tui relies on the inefficient Unix `ptrace` system call to control and read the memory of the migrating process. Also, Unix does not cleanly support the ability to transfer the kernel state and communication links associated with a migrating process. To gain maximum performance and functionality, Tui should be incorporated into an operating system that already supports homogeneous process migration.

9.3.4 Selecting Preemption Points

In the current implementation of Tui, preemption points are inserted at well known locations within the program. There still exist several open research questions with

regard to the accurate placement of these points.

Firstly, it is desirable to guarantee a maximum bound on the delay between initiating migration and actually halting the process at a preemption point. By placing points at frequent intervals within the program we can achieve this property, but at the expense of extra type information. An analytical study of this issue has already been performed [10], although an empirical study may help to provide more insight into an optimal solution.

Secondly, and more importantly, it is necessary to determine where *not* to place preemption points. When a procedure contains non-migratible code, or code that has been optimized in a machine dependent manner, avoiding preemption points will help to achieve correct migration. It would be useful to statically analyze programs to determine these locations.

9.3.5 Generation of Bridging Code

As discussed in section 6.3, special bridging code can be used to reverse the effects of machine dependent optimization. For each preemption point, there must be a means of determining whether the state of the program on the source machine is any different from the state presented at the same preemption point at the destination machine. If there is a difference, extra code must be executed before the state can be translated.

The exact form of this bridging code, and the methods used for determining the difference between the two program states is an open question.

9.3.6 Binary Translation

Tui has avoided the issue of translating the program code of a process by assuming that the program has been compiled for all machines. In a large networked environment (such as the Internet), it is not possible to make this assumption due to the wide range of architectures and the frequent inability to share the executable code

amongst all hosts.

The easiest solution would be to transmit the program's source code, or optimized intermediate code, to the destination machine. The Slim Binaries system [26] could be used as a means of storing program code in a machine independent form, and then converting the program to final machine code at load time. An extension would be to provide a generic method of translating an executable program back to an intermediate form, in the same way that Tui has done for the data component of a process.

9.3.7 Dealing with Operating System Differences

One of the major assumptions made in this thesis is that the operating system interface must be uniform across all machines. That is, migratible programs must see the same set of system calls and library procedures on both the source and destination machines. In reality, this assumption is often violated by programs that use conditional compilation techniques to select between code for various architectures.

The approach taken by Tui has been to provide a uniform set of library procedures to hide the differences in the underlying system calls. Although the four different machines supported by Tui are very similar (they all support UNIX system calls), a small amount of code is required to map the parameters passed by the migratible program to the parameters expected by the operating system. Because this mapping is hidden in the library code, exactly the same source code can be used on all architectures.

Tui has made several assumptions about the structure of library code. Firstly, the interface provided by the library must be identical across all machines. Secondly, there can be no preemption points placed within the library, since it may not be possible to determine equivalent points of execution when the code is machine specific. Finally, the library code must be stateless, or at least it must be possible to translate or regenerate the state when a program is migrated.

Future research in the area of operating system heterogeneity could lead to the relaxation of these assumptions and will allow migration between a more diverse set of architectures. For example, a program using the X-windows system will perform initial configuration of its data structures by determining how many colours and/or pixels are available on the display. If the program is migrated, the display characteristics may change, hence requiring that the internal data structures be restructed. Further research may suggest better techniques for designing X-windows clients or servers such that migration is more easily obtainable.

Bibliography

[1] Ada 95 Reference Manual (Language and Standard Libraries), Revised International Standard (ISO/IEC 8652:1995). available on the World Wide Web at: `http://www.adahome.com/rm95/`.

[2] Matrix Multiplication Package. available by anonymous ftp from: `usc.edu/pub/C-numanal/matmult.tar.gz`.

[3] The MicroEMACS Text Editor. available by anonymous ftp from: `ftp.agt.net/pub/Simtel/msdos/uemacs/ue312src.zip`.

[4] DWARF Debugging Information Format. Industry Review Draft, UNIX International, July 1993.

[5] Information Technology - Abstract Syntax Notation One (ASN.1) - Specification of Basic Notation. International Organization for Standardization, February 1994.

[6] Bruno Achauer. The DOWL Distributed Object Oriented Language. *Communications of the ACM*, 36(9):48, September 1993.

[7] Ali-Reza Adl-Tabatabai and Thomas Gross. Source-Level Debugging of Scalar Optimized Code. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation*, May 1996.

[8] Yeshayahu Artsy and Ralph Finkel. Designing a Process Migration Facility: The Charlotte Experience. *COMPUTER*, 22(9):47–56, September 1989.

[9] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[10] David G. Von Bank, Charles M. Shub, and Robert W. Sebesta. A Unified Modelor Pointwise Equivalence of Procedural Computations. *ACM Transactions on Programming Languages and Systems*, 16(6):1842–1874, November 1994.

[11] Micah Beck, James S. Plank, and Gerry Kingsley. Compiler-Assisted Checkpointing. Technical Report CS-94-269, University of Tennessee, Knoxville, December 1994.

[12] Matt Bishop, Mark Valence, and Leonard F. Wisniewski. Process Migration for Heterogeneous Distributed Systems. Technical Report PCS-TR95-264, Department of Computer Science, Dartmouth College, August 1995.

[13] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and Abstract Types in Emerald. *IEEE Transaction on Software Engineering*, 13(1):65–76, January 1987.

[14] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[15] Allan Bricker, Michael Litzkow, and Miron Livny. Condor Technical Summary. Technical report, Computer Sciences Department, University of Wisconsin-Madison, January 1992.

[16] David R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, 1988.

[17] William Clinger and Jonathan Rees (Editors). Revised (4) Report on the Algorithmic Language Scheme. *ACM LISP Pointers IV*, July 1991.

[18] Pure Atria Company. Purify: Fast detection of memory leaks and access errors. White paper available on the World Wide Web at: `http://www.pureatria.com/products/purify/fast_detection.html`, 1997.

[19] Max Copperman. Debugging Optimized Code Without Being Misled. *ACM Transactions on Programming Languages and Systems*, 16(3):387–427, May 1994.

[20] G. Coschi and J.B. Schueler. *WATFOR-77 – Language Reference.* WATCOM Publications Limited, 1989.

[21] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler Support for Garbage Collection in a Statically Typed Language. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 273–282, June 1992.

[22] Fred Douglis. Transparent Process Migration : Design Alternatives and the Sprite Implementation. *Software Practice and Experience*, 21(8):757–785, August 1991.

[23] Fred Douglis and Brian Marsh. The Workstation as a Waystation : Integrating Mobility into Computing Environments. *The Third Workshop on Workstation Operating Systems (IEEE)*, April 1992.

[24] F. Brent Dubach, Robert M. Rutherford, and Charles M. Shub. Process-Originated Migration in a Heterogeneous Environment. In *ACM Conference on Computer Science*. ACM. New York., 1989.

[25] D.L. Eager, E.D. Lazowska, and J. Zahorjan. The Limited Performance Benefits of Migrating Active Processes for Load Sharing. *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 63–72, May 1988.

[26] M. Franz and T. Kistler. Slim binaries. Technical report, Department of Information and Computer Science, University of California, Irvine, June 1996.

[27] Chris Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, 1995.

[28] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts*. John Wiley and Sons, 1987.

[29] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, May 1996.

[30] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1997.

[31] Mark Greenstreet. Synchronized Transitions: pre-draft of a language manual, 1996.

[32] John Hennessy. Symbolic Debugging of Optimized Code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, July 1982.

[33] H. Herlihy and B. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, October 1982.

[34] Urs Holzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. *ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, June 1992.

[35] Wilso C. Hsieh, Paul Wang, and William E. Weihl. Computation Migration: Enhancing Locality for Distributed Memory Parallel Systems. *SIGPLAN Notices*, 28(7):239–248, July 1993.

[36] Apple Computer Inc. MAE 3.0 White Paper. White paper available on the World Wide Web at: `http://www.mae.apple.com`, 1997.

[37] Kathleen Jensen and Niklaus Wirth. *Pascal - User Manual and Report - ISO Pascal Standard.* Springer-Verlag, 1985.

[38] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, February 1988.

[39] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 1988.

[40] Sanjay Krishnamurthy. A Brief Survey on Scheduling for Pipelined Processors. *SIGPLAN Notices*, 25(7):97–106, July 1990.

[41] Rodger Lea, Christian Jacquemot, and Eric Pillevesse. COOL : System Support for Distributed Programming. *Communications of the ACM*, 36(9):37–46, September 1993.

[42] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[43] General Magic. Telescript technology, 1996.

[44] Julia Menapace, Jim Kingdon, and David MacKenzie. The "stabs"Debug Format. Technical report, Cygnus support.

[45] Sun Microsystems. *Open Network Computer : RPC Programming.* The official documentation for Sun RPC and XDR.

[46] Dejan Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process Migration. *available on the World Wide Web at* `http://www.opengroup.org/~dejan/papers/indx.htm`.

[47] Herman Moons and Pierre Verbaeten. Object Migration in a Heterogeneous World - A Multi-Dimensional Affair. In *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*, pages 62–72, Asheville, North Carolina, December 1993.

[48] Mark Nuttall. A Brief Survey of Systems Providing Process of Object Migration Facilities. *Operating Systems Review*, page 64, October 1994.

[49] James S. Plank, Micah Beck, and Gerry Kingsley. Libckpt: Transparent Checkpointing under Unix. In *USENIX Technical Conference*, 1995.

[50] Steve Pope. Application Migration for Mobile Computers. In *3rd International Workshop on Services in Distributed and Networked Environments (SDNE 96)*, June 1996.

[51] Michael L. Powell and Barton P. Miller. Process Migration in DEMOS/MP. *Proceedings of the 9th Symposium on Operating System Principle*, October 1983.

[52] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald : A General-Purpose Programming Language. *Software Practice and Experience*, 21(1):91–118, January 1991.

[53] Rama N. Reddy and Carol A. Ziegler. *Fortran 77 with 90 - Applications for Scientists and Engineers.* West Publishing Company, 1994.

[54] Alexander B. Schill and Markus U. Mock. DCE++ : Distributed Object-Oriented System Support on top of OSF DCE. Technical report, Institute of Telematics. University of Karlsruhe, Germany.

[55] Ravi Sethi. *Programming Languages - Concepts and Constructs.* Addison Wesley, 1996.

[56] Charles M. Shub. Native Code Process-Originated Migration in a Heterogeneous Environment. In *ACM Conference on Computer Science.*, pages 266–270. ACM. New York., 1990.

[57] Gabriel M. Silberman and Kemal Ebcioglu. An Architectural Framework for Supporting Heterogeneous Instruction-Set Architectures. *IEEE Computer*, 26(6):39–56, June 1993.

[58] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary Translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[59] Daniel Dominic Sleator and Robert Endre Tarjan. Self-Adusting Binary Search Trees. *Journal of the ACM*, 32(3), July 1985.

[60] Peter Smith and Norman C. Hutchinson. Heterogeneous Process Migration : The Tui System. *To appear in Software - Practice and Experience*, 1997.

[61] Richard M. Stallman. Using and Porting GNU CC. 1995.

[62] Bjarne Steensgaard and Eric Jul. Object and Native Code Process Mobility Among Heterogeneous Computers. In *Symposium on Operating System Principles*, 1995.

[63] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, 1994.

[64] Volker Strumpen and Balkrishna Ramkumar. Portable Checkpointing and Recovery in Heterogeneous Environments. Technical Report ECE-96-6-1, Department of Electrical and Computer Engineering, University of Iowa, July 1996.

[65] Andrew S. Tanenbaum, Hans van Staveren, Ed G. Keizer, and Johan W. Stevenson. Description of a Machine Architecture for use with Block Structure Languages. Technical report, Vrije Universiteit Amsterdam, 1983.

[66] A.S. Tanenbaum, H. van Staveren, E.G. Keizer, and J.W. Stevenson. A Practical Toolkit for Making Portable Compilers. *Communications of the ACM*, 26(9):654–660, September 1983.

[67] M. M. Theimer and B. Hayes. Heterogeneous Process Migration by Recompilation. In Also available as Xerox PARC Technical Report CSL-92-3, editor, *11th International Conference on Distributed Computing Systems*, May 1991.

[68] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable Remote Execution Facilities for the V-System. In *Proceedings of the Tenth Symposium on Operating System Principles*, December 1985.

[69] Ian Toyn and Alan J. Dix. Efficient Binary Transfer of Pointer Structures. *Software - Practice and Experience*, 24(11):1001–1023, November 1994.

[70] Paul S. Wang. *An Introduction to Berkeley Unix.* Wadsworth Publishing Company, 1988.

[71] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. *Submitted to ACM Computing Surveys*, 1996.

[72] D. B. Wortman, S. Zhou, and S. Fink. Automating Data Conversion for Heterogeneous Distributed Shared Memory. *Software Practice and Experience*, 24(1):111–125, January 1994.

[73] Edward R. Zayas. Attacking the Process Migration Bottleneck. In *Symposium on Operating System Principles*, pages 13–22, Austin, TX, November 1987.

[74] Songnian Zhou, Michael Stumm, Kai Li, and David Wortman. Heterogeneous Distributed Shared Memory. *IEEE Transactions on Parallel and Distributed Systems*, page 540, September 1992.

Appendix A

Tui Manual Pages

A.1 The ack Command

NAME

ack — the Amsterdam Compiler Kit

SYNOPSIS

`ack -m`*machine* `-g -ansi (-tuibin | -tuilib)` *normal-arguments*

DESCRIPTION

The ACK ANSI-C compiler has been modified so that it generates executable programs suitable for migration by Tui. The compiler is able to generate code and type information for the four architectures currently supported by Tui. This manual page only shows the difference between the normal operation of the Amsterdam Compiler Kit and that required for creating Tui programs. See `ack(1)` for full details of the ACK compiler.

By default, executable programs are linked with the special Tui version of the ANSI-C library. This library contains migratible versions of the standard functions, as well as a set of UNIX compatible file and terminal I/O functions for accessing the Tui file server.

Once a program has been successfully compiled into executable format (using the `-o name` option), the following files will have been created:

- *name* – An executable file in the local operating system's format. This is the program that is actually executed.

- *name*`.ack` – An ACK format executable file that contains the same information as *name*, but is in a standard format that is consistent across operating systems. This is used by Tui when reconstructing the process.
- *name*`.tui` – A special file containing all the type information that is known about the program. This file is created by the `tuiprep` program that is implicitly called during the final phase of compilation.

OPTIONS

`-m`*machine*

Selects a compiler backend that will generate code for the architecture of your choice. The current choices are `sparc.mig` (for Sun 4 computers running Solaris), `sun3.mig` (Sun 3 computers running SunOS), `i386.mig` (Linux Machines) and `rs6000.mig` (PowerPC computers running AIX). Although other backends exist, only those with the `.mig` extension are capable of producing enough type information (that is, preemption points and call points) for Tui to function correctly.

`-g`

This option is required so that the compiler will generate sufficient type and location information about the data values in the program.

`-ansi`

This option is required. Currently, only programs that are written in ANSI C are migratible by Tui.

`-tuibin`

Generate preemption points and call points. These are required in order to halt/restart a process correctly. Failure to include this flag will result in an executable program that can't be halted at migration time.

`-tuilib`

Similar to `-tuibin`, but only call points are generated. This is typically used when compiling library code that shouldn't be interrupted by a preemption point. That is, some functions require code that is not migratible, therefore, excluding preemption points will not permit the program to stop during the execution of such a function.

A.2 The `migrout` Command

NAME

migrout — migrate a process from the source machine into an intermediate file.

SYNOPSIS

`migrout` *top-dir pid bin-file dump-file* [*debugging-options*]
[`stopat` *point*] [`args` *command-line-arguments*]

DESCRIPTION

`migrout` takes an active process, halts it, then converts its memory image into a machine independent intermediate representation that will be transmitted to the destination and used as input by the `migrin` command.

`migrout` depends on compiler generated type information to dissect the process image into its individual data components. These values are then recorded in an intermediate disk file in an architecture independent way, that is, no pointer values or register numbers are used. Finally, the source process is destroyed. Typically, the `migrin` program will be run on the destination machine, immediately following execution of `migrout`

PARAMETERS

top-dir

The directory under which the program's executable (`.ack`) file and type description (`.tui`) file are located.

pid

The UNIX process ID of the process that you wish to migrate.

bin-file

The basename of the executable file and type information file. The executable program is contained in the file:

top-dir/*arch*/tests/*bin-file*.ack

The program's type information is contained in the file:

top-dir/*arch*/tests/*bin-file*.tui

For both these files, *arch* must be one of: `sun4sol`, `sun3`, `i386` or `ibm`.

dump-file

The name of the intermediate file that will contain the machine independent representation of the program once `migrout` has completed. The complete file name of the intermediate file will be:

top-dir/dumps/*dump-file*

This file would typically be transmitted to the destination machine and supplied as input to the `migrin` program.

OPTIONS

`stopat` *point*

Instead of halting the program at the next preemption point that it reaches, only require it to stop at this user specified location. The feature is useful for debugging purposes only. It has the additional feature of causing `migrout` to execute the program for itself, rather than migrating an existing process specified by the *pid* argument. Preemption point numbers can be determined by using the `show_points` program.

`args` *command-line-arguments*

When this option is used in conjunction with the `stopat` option, `migrout` will execute the program with these command line arguments.

The following options don't affect Tui's operation, they simply request that `migrout` display information about what it is doing. This information is intended to aid in debugging Tui, rather than providing useful information to the user.

`scan`

Display information about the way the memory is being scanned in order to locate the data values.

`malloc`

During the scanning phase, display information about the heap blocks that are found within the program.

`emit`

Display the contents of Tui's internal *value table* that contains a list of all the values that were found during the scan phase.

`encode`

Display information about the values being encoded from the source machine's memory to the intermediate file. It will list all memory locations, data values, and position within the value table.

`mach`

Show all the machine dependent data values (such as register values, and stack frame locations) as they are being extracted from the process image.

`names`

Ask Tui to use full path names for each variable, rather than simply displaying a small identifier name. That is, show how a value was located, by tracing the variable names, array references and pointers.

`times`

Display wall-clock execution times for the various components of the `migrout` algorithm.

`savecore`

After checkpointing the process, and before scanning the memory image, make a disk copy of the process.

`usecore`

Instead of checkpointing the process specified by the *pid* argument, retrieve the memory image from a disk file created by the `savecore` option. This is useful when debugging Tui, so that it is not necessary to repeatedly execute the program, simply to obtain its memory image.

ERRORS

If migration is successful, `migrout` should execute silently and will have produced an intermediate disk file. If migration is not successful, the following errors may be reported on the standard output:

1. A pointer refers to memory that does not exist – This error occurs when the pointer refers to an area of memory that has not already been identified by scanning the text, data, heap or stack segments. Often the pointer has moved out of range due to pointer arithmetic, illegal pointer casting, or in many cases it is when a referred-to heap block has been deallocated. In this final case of dangling pointers, a warning is given and the pointer is set to NULL. In all other cases, a detailed report is given and the migration is aborted.

2. A pointer refers to legal data, but is not aligned – A pointer (of any type) refers to a data item that is already known, but it does not refer to the first byte of that object. In this case, there is no way of successfully migrating the pointer value, so an error and a detailed report is given.

3. Internal errors – There still exist a large variety of internal errors that could occur, but do not occur as frequently as the first two. These errors include:

    - This existence of non-migratible language features (for example, the union data type) that should have been filtered out by the compiler.
    - Type and location information that is incorrect, probably because the program needs to be recompiled so as to update the information.
    - Resource limitation errors (e.g. out of memory).

A.3 The `migrin` Command

NAME

migrin — reconstruct a process from the intermediate file and continue its execution.

SYNOPSIS

`migrin` *top-dir bin-file dump-file debugging-options*

DESCRIPTION

`migrin` takes an intermediate file that was produced by `migrout` and recreates the process from it. It relies on having access to the executable program (on disk) and all the type information that was used to create the intermediate file.

PARAMETERS

top-dir

The directory under which the program's executable file and type description file are located.

bin-file

The basename of the executable file and type information file. The executable program is contained in the file:

top-dir/*arch*/tests/*bin-file*.ack

The program's type information is contained in the file:

top-dir/*arch*/tests/*bin-file*.tui

For both these files, *arch* must be one of: `sun4sol`, `sun3`, `i386` or `ibm`.

dump-file

The name of the intermediate file that contains all the data values from the program. This file will be read by `migrin` in order to reconstruct the process. The complete file name of an intermediate file will be:

top-dir/dumps/*dump-file*

OPTIONS

The following options don't affect Tui's operation, they simply request that `migrin` display information about what it is doing. This information is intended to aid in debugging Tui, rather than providing useful information to the user.

`recon`

Display the data from the intermediate file as it is being examined and inserted into the memory of the newly created process.

`mach`

Show all the machine dependent data values (such as register values, and stack frame locations) as they are being inserted into the process image.

`times`

Display wall-clock execution times for the various components of the `migrin` algorithm.

ERRORS

Most of the errors reported by `migrin` are due to inconsistent type information. That is, when the type information generated by the compiler is different to the data presented in the intermediate file, `migrin` is unable to correctly reconstruct the process. If an error of this type occurs, recompiling the programs on all architectures will solve the problem.

The remainder of errors are due to resource limitations (e.g. not enough memory).

A.4 The prdump Command

NAME

prdump — display the contents of an intermediate file generated by migrout.

SYNOPSIS

`prdump` *dump-file*

DESCRIPTION

Reads the Tui dump file (specified by *dump-file*) and displays its contents in a human readable form. No effort is made to interpret the data or display any of the relationships between items. This is intended for debugging purposes only.

OPTIONS

None.

A.5 The `fileserv` Command

NAME

fileserv — Tui file server for migrating processes.

SYNOPSIS

```
fileserv
```

DESCRIPTION

A location independent file server for Tui programs. Since UNIX only supports the concept of local file descriptors, it is necessary to have an external file server for use with migratible programs. That is, when a program moves to another machine, it must still communicate with the original file server. All communication between the user programs and the server is done with TCP/IP.

When first started, `fileserv` binds itself to a well known TCP port (see TUI_PROC_SERVER_PORT in server.h) and listens for connections and the consequent requests. All requests and replies are transmitted as ASCII strings. TCP is used to ensure that no data is lost.

The server must be executed with a controlling terminal so that migratible programs may give screen output, i.e. file descriptors 0, 1 and 2 must exist.

For a client program to make use of the file server, the following function call must be made before any I/O is attempted:

`tuiFS_set_server(`*IP*`);`

Where *IP* is the 32-bit internet address of the machine that is running the server. Currently, the host machine must be running Solaris, since the client program uses the Solaris interface for file access. This simplification was made to avoid conversion between the various file I/O interfaces that exist.

COMMANDS

A request to the server is made by sending an ASCII string of the form: "*command argument*". By using the Tui library, the client program can simply use the standard UNIX function names. Alternatively the `migrate` program can be executed from the UNIX command line.

`fileserv` understands the following Solaris functions: `open`, `close`, `read`, `write`, `unlink`, `lseek`, `stat`, `fstat`, `chdir`, `getdents`, `getwd`, `tcgetattr`, `tcsetattr`. For further information on the semantics of these functions, see the appropriate Solaris man pages.

The two remaining commands are specific to the server:

- `ping` – Test whether the server is alive. If so, it will return the string `alive`.
- `quit` – Terminates the file server.

OPTIONS

None

EXAMPLE

To write the string "ABC" to the file "test", the following sequence of requests is made. To ensure that only ASCII characters are being transmitted (for ease of debugging) the string is first encoded into hexadecimal before it is sent.

```
open test 258  => 4        # create and open the file "foo"
write 4 4 41424300 => 4    # write the string "ABC"
close 4 => 0               # close the file
unlink test => 0           # delete the file
```

A.6 The procserv Command

NAME

procserv — a per machine process manager for Tui

SYNOPSIS

`procserv`

DESCRIPTION

To help manage migration, `procserv` provides an automated system for starting, stopping, killing and migrating processes. All these operations can be done without using the process server, but it is easier if you do use it, especially when dealing with processes on a remote machine. A single copy of `procserv` should be executed on each machine, then the standard migration commands should be sent to the server, rather than entered into a remote shell.

`procserv` uses a TCP/IP port (TUI_PROC_SERVER_PORT) for reliable communications. Requests can be sent, either by opening a connection to the server and sending ASCII commands, or by using the `migrate` command.

COMMANDS

`start` *top-dir bin-file*

Start a new process. The executable file is in *top-dir*/*arch*/`tests`/*bin-file*. The ID of the newly created process is returned as an ASCII string.

`migrout` *top-dir pid bin-file dump-file flags*

Execute the `migrout` command as if it was entered from the command line. See the manual page from `migrout` for more information.

`migrin` *top-dir bin-file dump-file flags*

Execute the `migrin` command as if it was entered from the command line. See the manual page from `migrin` for more information.

`pause` *pid*

Pause the process that has *pid* as its process ID.

`unpause` *pid*

Continue a process (with ID of *pid*) that has previously been paused.

`ping`

Test whether the process server is currently active. If it is, the return message will contain the architecture of the machine that is executing the server, and a list of the process IDs for all processes that the server is currently managing. This information can be used by client programs if necessary.

`quit`

Terminate the process server.

OPTIONS

None

A.7 The `migrate` Command

NAME

migrate — send a command to either a process server or a file server

SYNOPSIS

`migrate` *hostname* (`proc` | `file`) *command args*

DESCRIPTION

A command is sent to either the process server (`proc` option) or file server (`file` option) on the specified machine (*hostname*). The *command* and *args* are exactly those that are specified in the manual pages from `procserv` and `fileserv`.

Typically the `migrate` command would be called from a controlling script (e.g. Tcl/Tk or Perl), although it may also be useful when executed from the command line.

EXAMPLE

```
% migrate cascade proc ping          # check if the server is OK
alive sun4sol 21452 21480 21502
% migrate columbia file unlink foo   # delete a file called "foo"
0
```

A.8 The show_points command

NAME

show_points — display all preemption and call points in a program.

SYNOPSIS

`show_points` *executable-file*

DESCRIPTION

All the functions within *executable-file* are listed, along with the set of preemption points and call points within each function. This tool is used for debugging purposes when it is necessary to determine an exact preemption or call point number within a large program.

OPTIONS

None

A.9 The tuiprep Command

NAME

tuiprep — preprocess the compiler generated type information.

SYNOPSIS

tuiprep *executable-file output-file debugging-options*

DESCRIPTION

`tuiprep` will examine the debugging information that is generated by the ANSI-C compiler and will preprocess it into a form that is more efficient for Tui to use. The following operations will be performed:

- Type-checking is done across object files.
- The global variables are sorted into address order.
- The local variables for each function are sorted into frame pointer offset order.
- Information is converted from the standard *stabs* debugging format into the internal structure required for efficient access by Tui.

The type information contained within the *executable-file*, with a `.ack` extension, is used to create a new file with a `.tui` extension.

OPTIONS

`stabs`

Show the type information as the standard *stabs* format strings are being parsed. For debugging purposes only.

`points`

Display information about the location of the preemption points within the program.

`endian`

Generate a type information (`.tui`) file for use by a little endian machine (such as the i386 processor).

ERRORS

If a type inconsistency between two separate object files is located, an error is reported. Assuming that the compiler will always generate correct type information, no other errors can occur.

A.10 The tuiprepprint Command

NAME

tuiprepprint — display the type information from the preprocessed `.tui` file.

SYNOPSIS

`tuiprepprint` *tui-file options*

DESCRIPTION

Display the contents of a Tui information file (with a `.tui` extension). Typically this command would only be used by somebody who is interested in the fine details of a program's type structure. Most users would not wish to concern themselves with this level of detail.

The 6 sections of a `.tui` file are:

1. File header
2. Type information
3. Preemption points
4. Call points
5. Actual parameters (for each call point)
6. String table

OPTIONS

One or more of these options can be selected, depending on which of the file's sections are to be displayed.

`head`

Display the file header that contains the size (in number of elements) of each of the file's sections.

`type`

Display all the type information known about the program. This includes details of the functions, global variables, local variables and temporaries. The output from this option is usually very large.

`preempt`

Display the address of all the preemption points within the program. For a more descriptive list (showing function names), use the **show_points** program.

`call`

Display the program's call points, along with details of the arguments and temporaries that are alive at the call point.

`string`

List all the strings stored in the program's symbol table.

Appendix B

Making ANSI-C Programs Migratible

When attempting to use Tui to migrate a program written in ANSI-C, a series of steps must be followed to prepare the source code. Since Tui is not able to automatically migrate all programs, it is necessary to remove certain non-migratible language features that are present within the program. The following guidelines were followed when migrating the programs listed in section 4.1.

Making a program migratible is a process that has several stages. Firstly, the compiler will warn of the use of any non-migratible language features that can be detected statically. Once these have been removed, an attempt should be made to migrate the program at a well-known preemption point. Any migration-time errors that occur should be traced back to the offending location within the source code. Finally, performing this operation at all preemption points will most likely uncover all the non-migratible features that need to be removed.

B.1 Compiling the Software

Firstly, the source code for the program should be prepared for normal installation. It is assumed that it is written in portable ANSI-C, without any machine specific compilation (such as using #ifdef) or operating system dependencies. The Makefile should then be altered so that the standard compiler (normally `cc` or `gcc`) is replaced by `ack`. Finally, the software should now be built in the standard way (such as by typing `make`).

At this point the program should compile and execute correctly (but may not migrate). There will conceivably be a large number of a compilation warnings, but these can be ignored. Unless the program uses any non-standard variations on ANSI-C or contains references to any functions that are not part of Tui's standard ANSI-C library, there should be no compile errors.

B.2 Studying the Warnings

The next step to ensuring that the program will be migratible is to carefully study all the warnings that the compiler is giving. Many of these warnings were already reported by the compiler (such as parameter mismatches), whereas other warnings (such as use of unions) were added for use with Tui.

The program should now be modified to remove all of these warnings. Experience has shown that the majority (up to 90%) are due to the lack of the correct ANSI-C prototypes or function headings. These can easily be written by hand or by a tool such as `protoize`.

The remaining warnings must be considered in more detail, since each will cause a different amount of concern. If it is clear that a particular warning is due to a non-migratible feature, the code must be rewritten to avoid use of that feature. Often this may be a small change to one line of the code, but it is also possible that single warning can lead to major reconstruction of the program.

B.3 Migrating From a Specific Preemption Point

Once all the compile-time warnings have been removed, we now attempt to migrate the program in a controlled way. Stopping the program at a single preemption point, in a well-known location, will sometimes result in a series of migration-time errors that must be fixed. For the example programs, a suitable preemption point was chosen by selecting a point that followed the program's initialization functions.

A migration-time error could either report that a pointer refers to an unknown area of memory, or that a pointer refers to a location within a data value that is known to be indivisible. In both cases, the error report contains the numeric address of the data (or the nearest known data value), a description of the type of the data, and the textual name of the data values (containing details of any pointers that were followed to locate the data).

After a migration-time error is reported, the programmer must examine the source code to determine why the data values are causing the problem. Due to Tui's construction, it is always the case that one of the conflicting values is either a global variable, a local variable, or a heap block, and the other value will be a pointer. By determining how the pointer variable came to incorrectly point to (or near) the memory location, the source code can be modified to avoid use of the non-migratible feature.

Given that the name and scope of the data values is well known, a programmer who has experience in making programs migratible can often locate and possibly correct a problem in several minutes. If no migration-time errors are reported, then the program will be ready to start executing on the destination machine.

B.4 Fixing Runtime Bugs

Once the program is executing on the destination machine, there is a possibility that the program will not continue to function correctly. This either occurs when the

migration tool was unable to extract all the necessary data values from the program, or if the migration tool itself contains a bug. In previous experience, the majority of these problems were due to bugs in Tui that have now been fixed.

The only non-migratible feature that can only be detected at run-time is when pointer arithmetic causes a pointer value to refer to a valid area of memory that is not within the pointer's intended range. For example, if a pointer originally refers to one heap block, but is incremented beyond the end of that block so that it now refers to a second block, the pointer will be migrated relative to the second (incorrect) block. Although this problem is possible, it has not yet materialized in practice, as any pointer that refers outside its intended memory area is considered to be undefined and should no longer be used.

Detecting a bug at run-time can be quite difficult. Firstly, the method for detecting whether a program has successfully continued or not, will vary between programs. For the ST compiler, a comparison was made between the correct output and the migrated program's output. For MicroEMACS, a buffer was loaded into the editor and later examined for consistency. The bc calculator was migrated during the calculation of a complex formula, and the answer was compared with the expected answer.

B.5 Exhaustive Testing

The final step of ensuring that a program is migratible is to attempt migration at all preemption points. This is necessary since the amount of context being translated will differ between functions. That is, local variables and localized variations in global and heap values may cause the program to be non-migratible. An automated script can be used to migrate the program at each preemption point and test whether the migration was successful.

A limitation of this method of migrating at each preemption point is that migration will only occur the first time that point is reached. To avoid this situation,

a limited amount of delayed preempting should be performed. That is, the program should be permitted to execute for a few seconds before migration is initiated.

It could be argued that these testing methods will not guarantee that we locate all the non-migratible features. Unless we test every possible path through the program with every possible input, we can never before sure that migration will always be successful. Since test coverage is beyond the scope of this thesis, the testing methods that have been used are considered satisfactory.