Heterogeneous Process Migration : The Tui System

Peter Smith and Norman C. Hutchinson,
Department of Computer Science
University of British Columbia
Vancouver, B.C., V6T 1Z4,
Canada
email: {psmith,norm}@cs.ubc.ca

March 14, 1997

Abstract

Heterogeneous Process Migration is a technique whereby an active process is moved from one machine to another. It must then continue normal execution and communication. The source and destination processors can have a different architecture, that is, different instruction sets and data formats. Because of this heterogeneity, the entire process memory image must be translated during the migration.

"Tui" is a migration system that is able to translate the memory image of a program (written in ANSI-C) between four common architectures (m68000, SPARC, i486 and PowerPC). This requires detailed knowledge of all data types and variables used with the program. This is not always possible in non-type-safe (but popular) languages such as ANSI-C, Pascal and Fortran.

The important features of the Tui algorithm are discussed in great detail. This includes the method by which a program's entire set of data values can be located, and eventually reconstructed on the target processor. Performance figures demonstrating the viability of using Tui to migrate real applications are given.

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, so that no changes need to be made to the memory image.

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

This paper examines an experimental Heterogeneous Migration system known as *Tui*. An implementation has revealed the issues involved in translating the data component of a migrating process. Tui does not address the issues normally associated with homogeneous migration, nor does it address the translation of a program's instructions between different architectures.

2 Motivations

The traditional reasons for using process migration have been identified [1] 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 [2]. 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, the 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 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 be 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 mathematics 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 of 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 the mainstream platforms (such as MSDOS, MS 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.

2.1 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, normally done via wireless LANs or cellular telephones.

It has been proposed [3] 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 potentially follow them.

2.2 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 [4]. 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.

Wide area processing is a topic that has already been addressed in the Java [5] and Telescript [6] 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.

3 Heterogeneous Migration and the Tui System

3.1 Existing Systems

Before discussing the purpose 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. Further references are given in section 7.

Heterogeneous migration systems can be classified into the following categories:

1. **Passive object –** The process (or object) 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 –** 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 –** 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** – 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 should be considered when designing a heterogeneous migration system 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 in a known state will reduce the amount of information to be migrated, and simplify 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 known to the authors 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 become more difficult or even impossible.

3.2 The Purpose of Tui

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, Pascal, 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 becomes more 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 and assign a type to 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.

- The programmer must refrain from using the union type.

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:

1. 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. Other systems (such as UNIX) will have limited functionality.

2. 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, even though it was previously considered to be non-migratible.

3. 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.

4. 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.

5. 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.

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

Ideally, the final migration algorithm, in conjunction with the language compiler, must be able to successfully migrate a process with no extra intervention from the programmer. That is, any existing software should be migratible without the need to alter the source code. If this is not possible, the compiler or migration system must warn the programmer of features in the program that are not migratible. As a last resort, a written document will describe any non-migratible language features that are not detectable by the compiler.

The eventual aim is to have a complete system where the programmer can take their existing programs and use Tui to migrate their software, or use Tui's suggestions to improve its migratibility.

4 The Tui Migration Algorithm

In its current form, the "Tui Heterogeneous Process Migration System" is able to migrate *type-safe* 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. A program is considered to be type-safe if it is possible to uniquely determine the type of each data value within the program. Some work has been done to relax the type-safety restriction, although this is mostly left as future research.

The Tui software has existed in both a prototype and final form. It is important to note that several lessons were learned from the prototype, and lead to the creation of the final implementation. The prototype version made use of a garbage collection style algorithm for locating blocks of data to be migrated. Practical studies and performance tests indicated that this method was not sufficient in many situations, and hence motivated a second version.

This section 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 section only discusses the second version, a comparison between the two systems will be made in section 5.

4.1 Overview of Tui

Figure 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 ma-

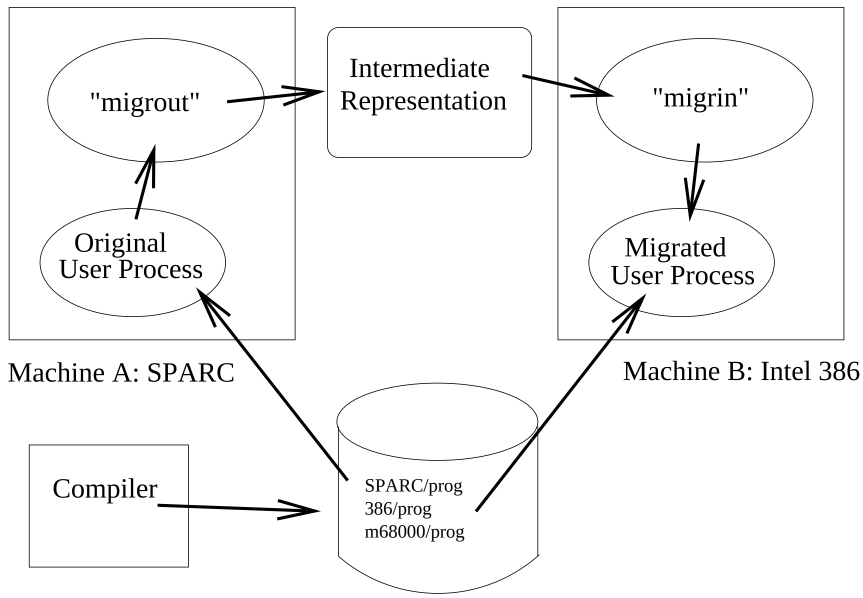

Figure 1: The Tui Migration System

chine, 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) [7] 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 fetch the memory of the process and scan the global variables, stack and heap to locate all data values. Finally, all these values are converted into an intermediate form and sent to the target machine.

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 com-

piled 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 the 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 type-safe.

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.

4.2 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 [8] 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 [9] 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)[7]. This system is very easy to modify, and contains source code for all components. It has frontends for languages such as 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.

4.2.1 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 has exactly the same structure on all target machines. That is, each program is compiled to contain 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 [10], 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 [11] are not relevant here.

ACK front ends generate *stabs* format [12] 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.

4.2.2 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 VAX 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.

4.3 "Migrout" : Checkpointing the Process

The following description of 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 its memory image is copied into Tui's address space for easy access. Thirdly, the type information is used as a guide for scanning this memory, and locating all data values. Finally, these values are translated into an intermediate format for transmission to the `migrin` component of Tui.

4.3.1 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 trees*, 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 2 shows the ASCII stabs strings for the given set of C declarations. It then shows the corresponding type tree entries.

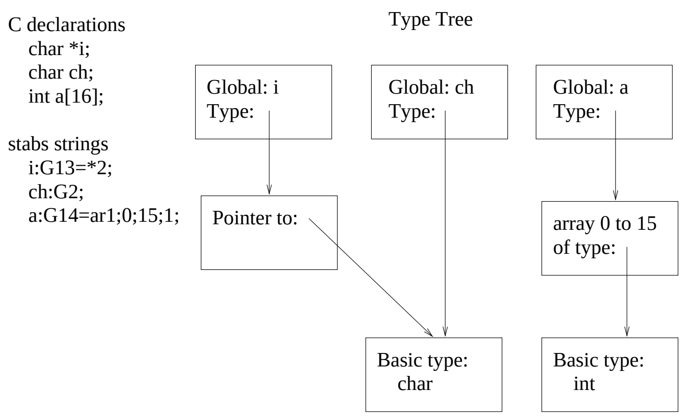

Figure 2: stabs strings and the type tree

In addition, two extra tables are required. The first table records the preemption 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 machine independent representation of the point's address.

4.3.2 Halting the Process

Halting a program is more complex than in homogeneous migration. 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 one breakpoint, but it is not always easy to determine which preemption point will be reached next.

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.

4.3.3 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 deal with 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 5.

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, the absolute addresses must be calculated. Special care is also taken to maintain a correct idea of the current register set, especially since they 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.

4.3.4 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 4.5 gives full details of the intermediate file format.

The only difficulty of 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`.

4.4 "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 therefore can mostly be 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.

4.5 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 yet 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.

One interesting optimization has been made to the way in which integers are encoded. The number of bytes used to represent integers depends entirely on the value of the number, and not the size that it had on the source machine, or will have on the destination machine. That is, small values (such as 5) can be encoded in one byte, whereas larger values require more.

5 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 was not satisfactory. These limitations will now be examined in more detail.

5.1 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 intend to use 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 point `b` suggests that this same area of memory stores characters.

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

When this program is 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 a 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 has 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 malloc 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 will incorporate this size and type into a call to a special version of `malloc` that will record the information for later use by Tui.

Now that 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. It is expected that these changes will greatly expand the number of programs that are migratible. However, future work is needed to prove this.

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.

5.2 Performance

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 [13] 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 new version would give better performance for many programs.

The revised scanning algorithm will always give linear performance (based on the number of data items), 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.

6 Performance Tests

To fully test the performance of the Tui algorithms, three different test programs (for Tui to migrate) have been created. Each is 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 scan the heap space in a random order.
- `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.

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

To demonstrate that Tui can correctly function on the four supported architectures, each of the programs was migrated. Figures 3 to 6 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 analyzed, `fibonacci` and `arrays` were similar.

The exact machines are:

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

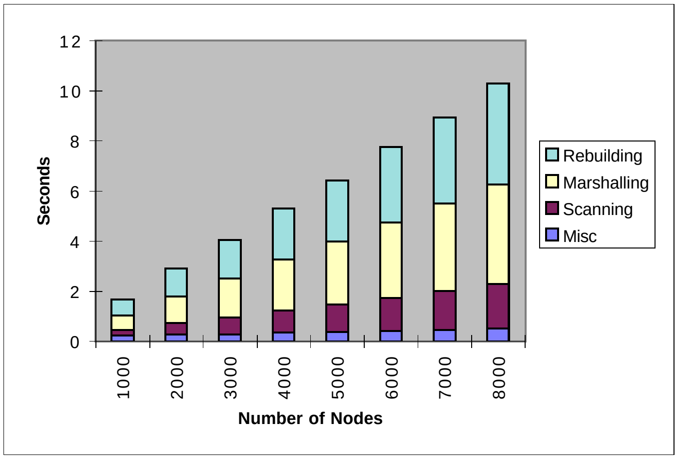

Figure 3: “tree” on Sun 4

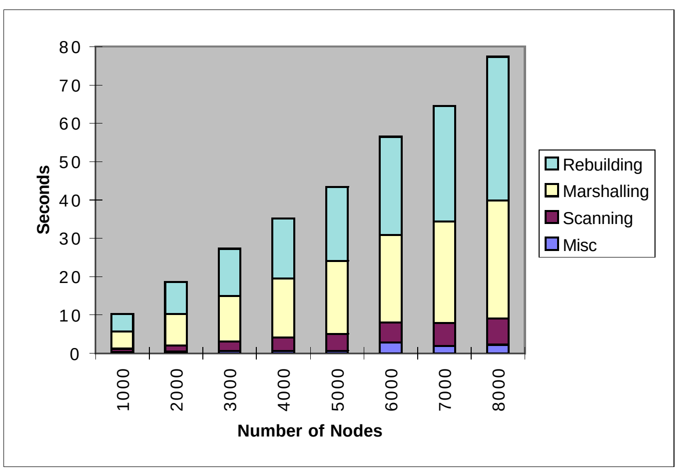

Figure 4: “tree” on Sun 3

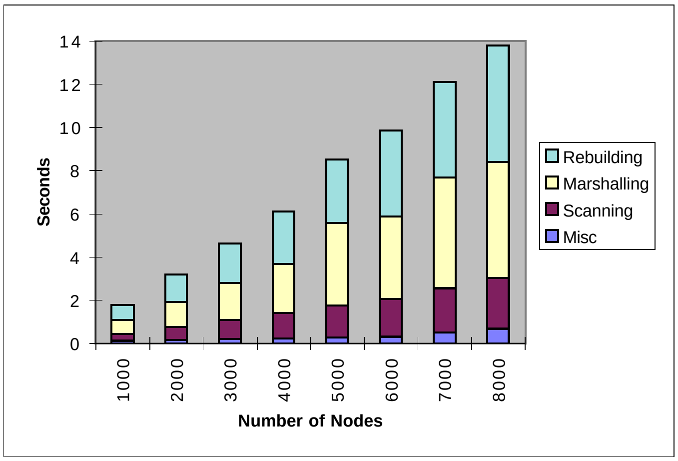

Figure 5: “tree” on i486

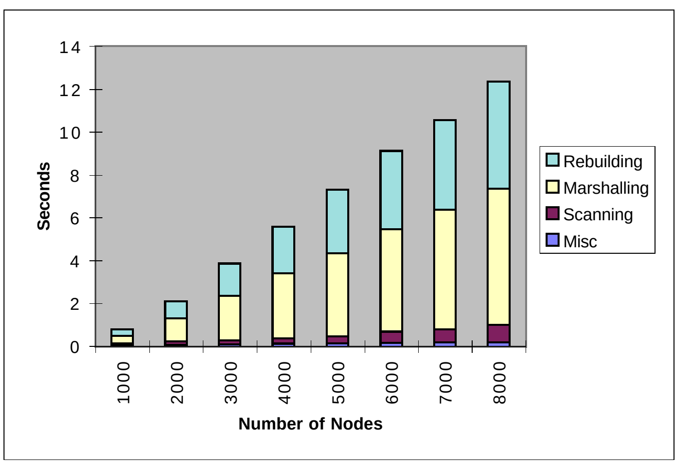

Figure 6: “tree” on PowerPC

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 size of the program, but the amount of type information will remain constant.

- `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.

6.2 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 7 to 9 show the contribution of the major costs (scanning, encoding and rebuilding) for various

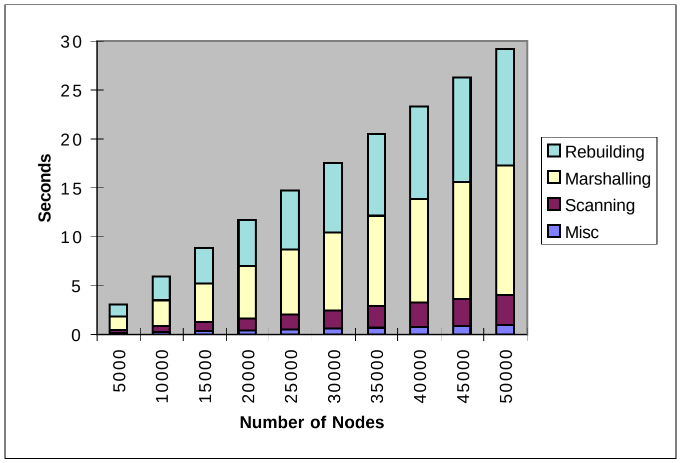

Figure 7: Growth of "tree"

input sizes. The following list gives an explanation of the performance for each of the three programs. To avoid 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 a slightly higher 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.

6.3 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

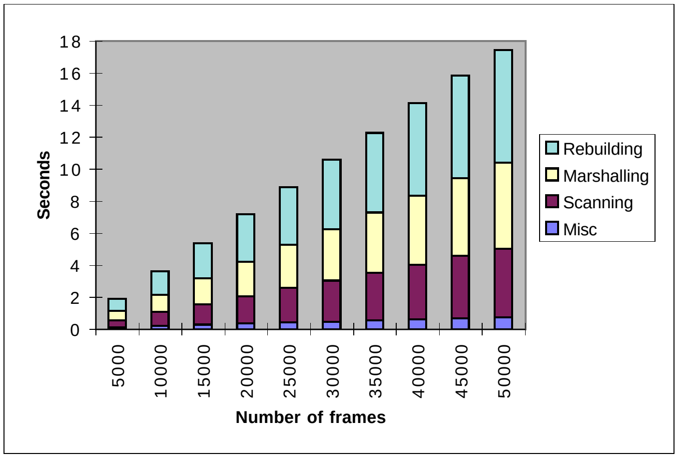

Figure 8: Growth of “fibonacci”

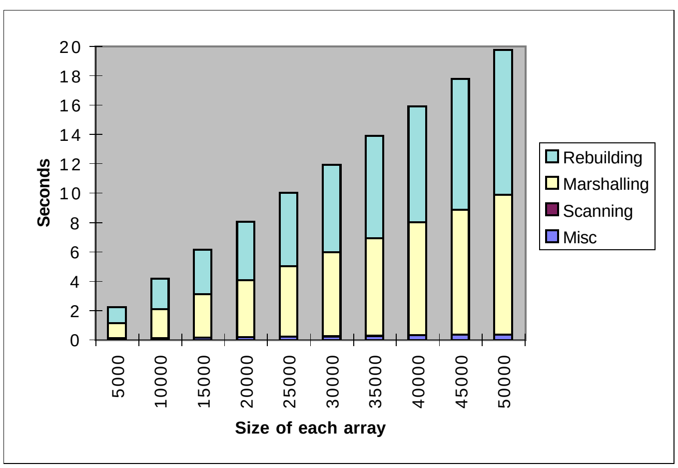

Figure 9: Growth of “arrays”

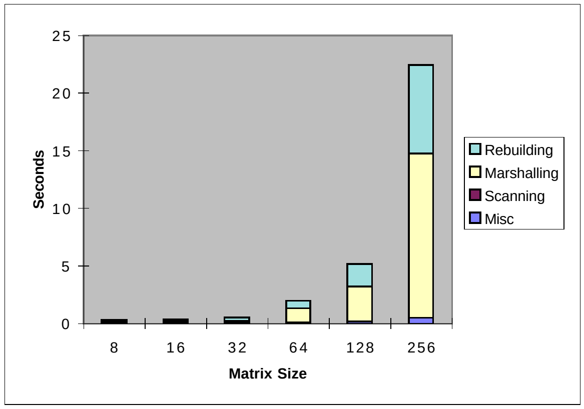

Figure 10: Growth of "matrix"

small set of contrived programs, two applications that were not designed for migration have been used. A matrix multiplication package and a popular text editor demonstrate how easily "real" programs can be migrated.

Matrix Multiplication

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 [14] contains approximately 1000 lines of code written in ANSI-C. Other than altering the program's main loop (to allow for user selected matrix sizes), no modifications to the code were necessary. This is clearly a desirable feature as our goal is that migration should be transparent to the programmer.

In figure 10, 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.

The MicroEMACS Editor

The second realistic program is a cut-down version of the popular EMACS editor, called MicroEMACS [15]. This C program contains approximately 20,000 lines of code (that

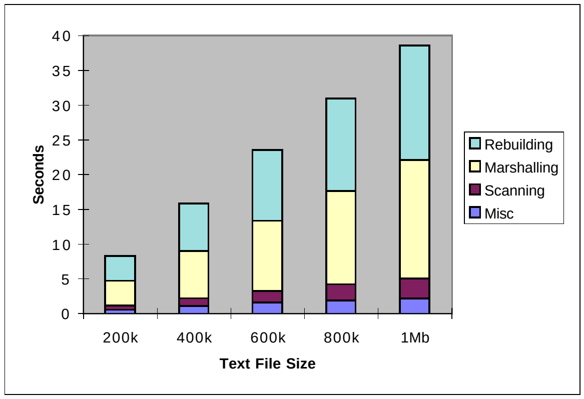

Figure 11: Growth of “uemacs”

were relevant when running under UNIX). Figure 11 shows the time required to migrate the editor while it was editing a text file. The file size ranged from 200Kb to 1Mb.

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. For this purpose, a transfer delay of up to a minute would be acceptable.

Unlike the matrix program, several minor changes were required. Firstly, the program was converted into ANSI-C by altering the function headings and adding function prototypes. Secondly, there were 10 uses of the “malloc” function that were missing the “sizeof” operator. In all cases, the compiler warned about the missing type information. Finally, there were 3 additional situations where a heap block was being allocated as a variable sized structure. Each of these occurences was fixed by using the technique discussed in section 5 (that is, adding a second call to “malloc”).

7 Related Work

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

This section 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 the range of approaches, in particular, how they compare to Tui. Finally, other supporting areas such as garbage collection and debugging will be surveyed.

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][17], Charlotte [18], DEMOS/MP [19], Sprite [20], Condor [21] and Accent [22]. A good summary of these and other systems is given in [23] as well as a more recent survey in [24]

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 [25] [1] [26], DOWL [27], DCE++ [28], COMET [29], and COOL [30].

Other heterogeneous migration systems

All of the following systems supports 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 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 [31][32] 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, but simpler than, Tui's type tree.

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) [33] 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 program's code. Placement of preemption points in Tui could benefit from this type of analysis.

A second system [34], 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 [35] 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 [25] [1] [11] 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 correct migration will always be possible.

The HMF (Heterogeneous Migration Facility) system [36] 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.

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 [21] [37], but they only function in a homogeneous environment. The recent concept of "Memory Exclusion" [38] demonstrates that careful selection of data values to be saved can reduce the cost of checkpointing. Another system [39] 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.

Debugging

Source level debugging is also closely related to heterogeneous process migration. Compilers generate extensive amounts of information describing such things as the type and location of all variables, and the location of each source code statement. A debugger such as `dbx` or `gdb` uses this information to aid the programmer in studying a process.

Recently, work has progressed in the field of debugging optimized code [40] [41]. Whereas traditional debuggers have only been able to correctly debug unoptimized programs, it has been recognized that many errors do not become obvious in this situation. The *DWARF* debugging format [42] is a newly developed format that is capable of expressing the structure of an optimized program.

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 [43] and ISO's ASN.1 [44].

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

One solution [45] 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 has also been solved by Tui through the use of the value table and the method of encoding pointers.

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 [46].

The 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 [47] 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, [48] 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 [49], 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 [50] [51] 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 will 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 [52]. There was a desire to convert existing VAX software to the Alpha platform, without using the original source code. Another system [53] talks about emulating complex instruction set machines by using binary translation within a RISC environment.

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.

8 Ongoing Work

Even though the current implementation of Tui has been successful, the following topics are seen as necessary additions to the work.

- A more detailed survey of non-migratible languages features will be made. We have already shown that for a program to be successfully migrated, type-safety is not always a requirement. Since we are not totally sure of the extent of "migratibility", a survey of common programs (for example, compilers, editors and scientific software) will be made. This work will act as a guide for those who wish to use Tui to migrate their own programs.

- The current implementation of Tui will be extended to deal with a wider range of programming languages. ACK already supports Pascal, Modula-2 and Fortran, so the coding requirements should be minimal. Since C is generally considered to be one of the least type-safe languages, it is expected that other languages will present fewer problems for migration.

- Realistic performance tests will be performed to demonstrate that migration is beneficial in wide area and mobile computing. A single program will be used to compare the cost of data access across the internet (while keeping the program at a fixed location), versus the cost of migrating that program closer to the data.

9 Summary

The Tui Heterogeneous Process Migration system is able to move a process between machines of different architecture, with minimal programmer intervention. It uses type information that is generated when the migratible program is compiled. The bulk of the algorithm's work involves locating and determining the type of each data value within the process. This data is marshalled into an intermediate form so that the process may be reconstructed on the destination machine.

The algorithms for checkpointing and reconstructing a process image have been described, with most focus placed on those features that are unique to heterogeneous migration. Performance measurements of several programs, including two real applications, have shown that migration is possible within an acceptable amount of time. This is especially true if we consider the potential cost of not migrating a program across a wide area or mobile link.

Tui has been designed to migrate ANSI-C programs, although other languages such as Pascal and Fortran could be supported. The required compiler changes are fairly minimal, so new languages can easily be incorporated into the system. A common problem is that most practical languages are not type-safe, so determining the correct type of each data item, therefore migrating successfully, is not always possible.

Experience with two versions of Tui has shown that some type-unsafe programs can be migrated, as long as the programmer is required to be more specific about the type details of dynamically allocated memory. Proposed future work will involve categorizing non-migratible language features, and attempting to eradicate them.

References

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

[2] 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.

[3] 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.

[4] 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.

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

[6] General Magic. Telescript technology, 1996.

[7] 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.

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

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

[10] 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.

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

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

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

[14] Matrix multiplication code. available by anonymous ftp from: usc.edu/pub/C-numanal/matmult.tar.gz.

[15] The microemacs editor. available by anonymous ftp from: ftp.agt.net/pub/Simtel/msdos/uemacs/ue312src.zip.

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

[17] 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.

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

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

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

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

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

[23] Mark Nuttall. A brief survey of systems providing process of object migration facilities. *Operating Systems Review,* page 64, October 1994.

[24] Dejan Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *http://www.opengroup.org/ dejan/papers/indx.htm*.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

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

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

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

[41] 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.

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

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

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

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

[46] Paul R. Wilson. Uniprocessor garbage collection techniques. *Submitted to ACM Computing Surveys*, 1996.

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

[48] 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.

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

[50] 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.

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

[52] 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.

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