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

February 28, 1996

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 prototype 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 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. Initial performance figures demonstrating the viability of using Tui for real migration 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 by 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 will 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*. A prototype 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.

The remainder of this paper is set out as follows. Section 2 discusses why process migration is an important feature in a modern operating system. Section 3 discusses existing heterogeneous migration systems and explains how Tui is different. The majority of the paper, in section 4, describes the details of the Tui migration algorithm. Section 5 shows the results of some performance tests. Finally, section 6 lists some proposed improvements to the current prototype implementation, and section 7 discusses some related work.

2 Motivations

The traditional reasons for using process migration have been identified [21] 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 [14]. 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). This improvement in performance can be significant.

- 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 this change 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 will 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 motivation 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 [13] 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 [20]. 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 a lot of sense in the case where the program is smaller than the data.

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 divided into these 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.

3.2 Emerald

The Emerald system [5] [21] [28] is an object oriented programming language and runtime system that supports movement of processes (active objects) between machines of different architectures (the category of active objects with native code). Emerald provides each object with a uniform view of the outside world, regardless of where is it located. Consequently, operating system heterogeneity is no longer a problem.

To obtain a version of the machine code that can be executed on the destination machine, there must exist a precompiled version of the object for each target architecture. When migration takes place, the machine code is loaded directly into memory. This is by far the simplest solution.

The most complex component of the migration algorithm involves locating and determining the type of the data. When an object is moved, its instance data (global data for that object) and the local data (generated by procedure calls) are retrieved from memory. They are then packaged and converted so they can be reinstated on the destination machine.

Emerald is a type-safe language where each data value is considered to be an object, and can be referenced by the use of an object ID (rather than requiring the use of pointers). Consequently, locating the data and determining its type is straightforward, so long as the necessary type information is generated by the Emerald compiler.

3.3 Purpose of Tui

Although the Emerald migration system works well, the primary limitation is in the popularity of the language. By far the majority of existing software, and programmer experience, is in older 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 the Emerald system.

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.

Our aim is to discover and attempt to solve the problems that make common languages unattractive for heterogeneous migration. Clearly is not possible to migrate all C programs, without sacrificing performance (if we were to tag all data values, migration would always be possible, although very inefficient). Tui will enable us to determine the possibilities and limitations of heterogeneous migration, to the extent of identifying how much of the migration process can be automated, and what limitations are placed on the programmer's coding style.

Ideally, the migration algorithm, in conjunction with the language compiler, is 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

The "Tui Heterogeneous Process Migration System" exists in a prototype form. It 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.

This section gives a complete description of the 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.

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 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) [36] 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 written to a file on disk.

4. On the destination machine, the `migrin` program takes the intermediate file and creates a new process. It is assumed that the program has been compiled for the

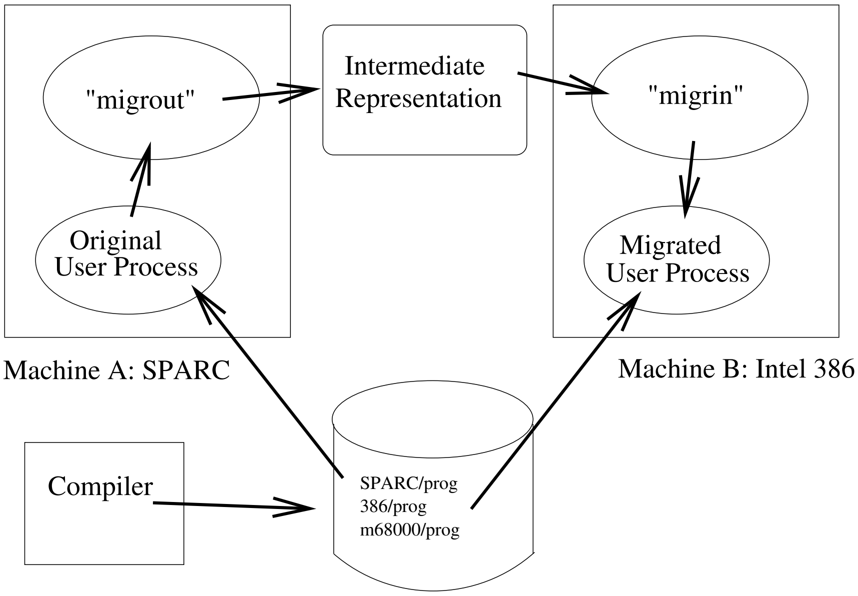

Figure 1: The Tui Migration System

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.

Most variants of Unix do not allow migration, so movement of communication links and files (other than stdin, 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 [33] 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 [15] 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 (The Amsterdam Compiler Kit). 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 [35], 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 [34] are not relevant here.

ACK front ends generate *stabs* format [23] 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 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.

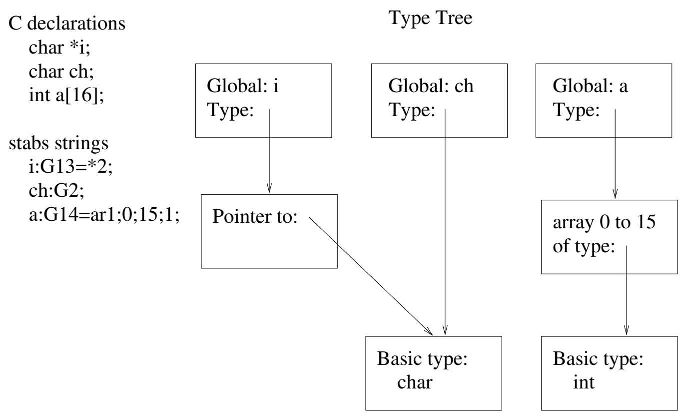

Figure 2: stabs strings and the type tree

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.

4.3.3 Scanning the memory

While searching the memory of the process, in an effort 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. Eventually, all useful data values will be known, and any type inconsistencies can be reported. For example, if a particular word in memory is thought to contain an integer valued global variable, but a pointer reference to that memory suggests that it contains a float.

Firstly, the 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. Therefore, no type consistency problems can arise.

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. It is most likely that the arguments will be scanned twice, once from the caller's perspective and once as the parameters of the callee's frame. As previously mentioned, this is necessary for procedures that allow a variable number of arguments. Also, it can help to ensure consistency between caller and callee.

To locate data values in the heap, Tui follows any pointers that were found in either the global or local variables (in a similar fashion to a garbage collector). At this point, we must determine whether the data being pointed to is already in the value table. If so, the type of the existing data must match the base type of the pointer. This process is continued recursively until pointers have been followed.

The value table itself has many features worth discussing. Insertions and lookups can be random, since pointers may refer to any memory location. Also, we often wish to locate the adjacent entries, as well as to make an *inorder* traversal of the entire table. A splay tree [32] has been used for this purpose.

When a new data value is discovered, it is important to consider any data that already exists at the same memory address, and any data that precedes or follows this new value. The following list of cases has been noted. In each case, Tui must check and report any type inconsistencies, and must be able to detect when two data values are in fact the same item, or one is part of another.

- If there is already some data at the address, we normally expect it to have the same type as the new data, otherwise the process could not have been type safe. For example, this fragment of code is non migratible since it violates this property.

```
{
    int a;
    char *b = &a;
}
```

- If the data that precedes the new item overlaps with it, then the new item must be a subelement of the old item. That is, if the base address of the newly discovered data item falls within the range of addresses already covered by an existing item. For example,

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

  Since `a` is entered into the table first, `*b` will be consumed by `a`.

- If the new data item consumes the successive data item, then the new item is entered into the table, and the old entry will be discarded. This would happen in the previous code fragment, if `*b` was scanned before `a`.

- In some special cases, it is possible that two data items are located at the same address, but have different types. For example,

```
{
    struct {
        int a;
        char b;
    } st;

    int *p = &st.a;
}
```

In this case, one data item is the whole struct, and the other is the element `st.a`. If the two types can be merged correctly, then largest of the two items will remain in the value table.

4.3.4 Translating 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. 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. During the scanning phase, the location of all these references was recorded in a table.

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. This is similar to the idea of using object identifiers in a language like Emerald.

The interesting problem that does not arise in Emerald is that pointers may refer to a subelement of a composite data item. If this sharing relationship is not preserved, the subelement will be encoded twice, and there would be no way for the destination machine to restore the data structure to its original state. For example:

```
{
    char buffer[10];
    char *p = &buffer[4];
}
```

In Tui, pointers are encoded as a pair of numbers (*Object ID*, *offset*), where the *offset* states how many indivisible subelements must be skipped in order to locate the correct value. In the above example, the *Object ID* would be whichever unique number was associated with the `buffer` array, and *offset* would be 4.

We must also consider the case of composite data items, that contain other composite data. For example,

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

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

In this case, the offset for `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 is less complex than for `migrout` since the intermediate file can be read sequentially. There is no need to discover the location of data items, therefore a value table is not used.

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. Although the ordering of data items on the stack is vital, it is not so important for heap values. However, it is convenient for `migrin` to keep data values in the order that they appeared in the source process.

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 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.

Because of this, a program is not required to have the same size for each data type across all machines (however, the current 4 processors do have this property). When `migrin` reads a data value from the intermediate file, it then decides whether the target storage location is large enough to hold that value. If so, the program can be migrated. For example, if the source machine uses 4 byte integers, but the destination machine has 2 byte integers, then a program can still be migrated as long as all values are small enough to fit into 2 bytes. This has the advantage of allowing more programs to migrate, however, it may cause failure at runtime.

5 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. Since the implementation is currently in prototype form, the absolute running times are not as important as the relative growth in complexity that occurs when the program size increases. These tests help to identify the components of Tui that will require the most optimization.

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. 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 large character arrays are dynamically allocated on the heap and then filled with characters. This test demonstrates the efficiency of encoding and reconstructing extremely large areas of memory.

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

To demonstrate that Tui can correctly function on the four supported architectures, the `tree` program was migrated. Figures 3 to 6 show the time taken by each of the main components of both the `migrin` and `migrout` algorithms. In these tests, the number of tree nodes varies from 100 to 6400, doubling in each step. Therefore, if the cost of migration doubles from one problem size to the next, the complexity is linear. Although only the `tree` program is analyzed, `fibonacci` and `arrays` will be similar, given the small input sizes.

The exact machines are:

- 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.

- `Symbols` – The time required to convert the debugging information (in stabs format) into the internal type structures. This is dependent on the amount of type information for the program, and not on the size of the process being migrated. It is therefore constant for all input sizes.
- `Reading core` – The time required to copy the data and stack segments from the migrated process into Tui's address space. This is dependent on the underlying operating system's implementation of the `ptrace` system call. It exhibits a small cost that grows slightly as the process size increases.
- `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.

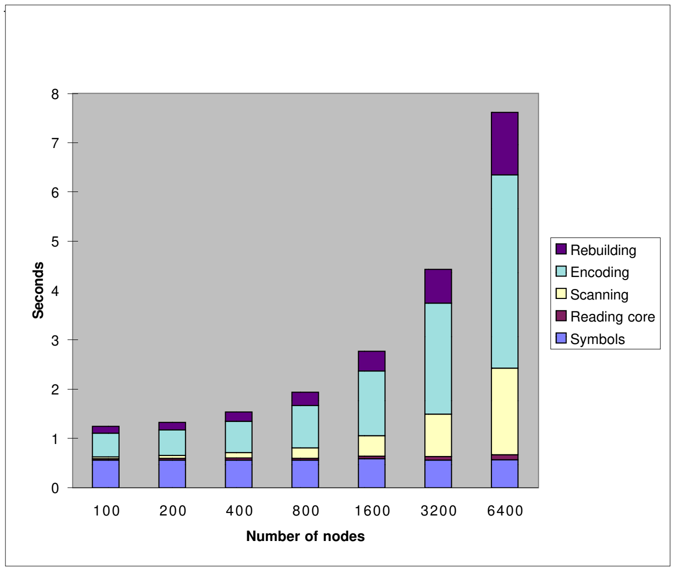

Figure 3: "tree" on Sun 4

- `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 building type structures and reading/writing core memory is the same as for `migrout`.

It can be seen that scanning, encoding and rebuilding are the major components of the migration cost. The cost of reading symbols is constant for a given program, and the cost of reading the process core is small enough to ignore. We next study how the cost of the three major components increases as the input size becomes extremely large.

5.2 Asymptotic Growth in Migration time

To examine Tui's performance when migrating realistic 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 input sizes. Note that the input size axis is linear and also that figure 8 has a truncated range due to its bad performance on large input sizes. The following list gives an explanation

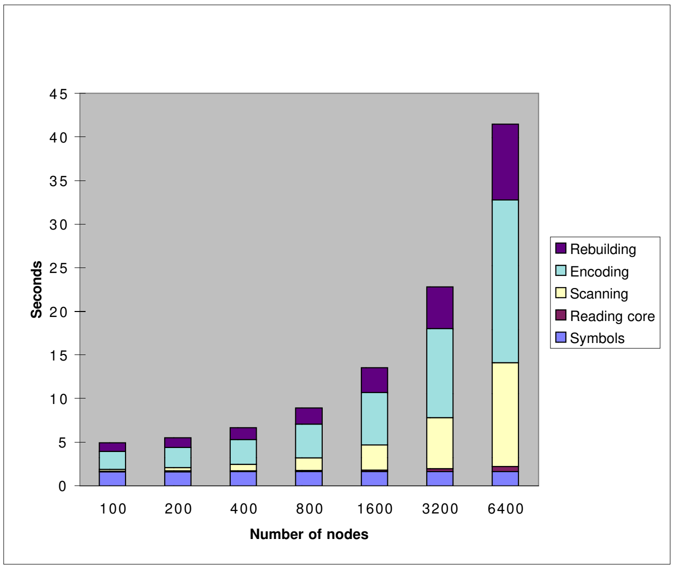

Figure 4: “tree” on Sun 3

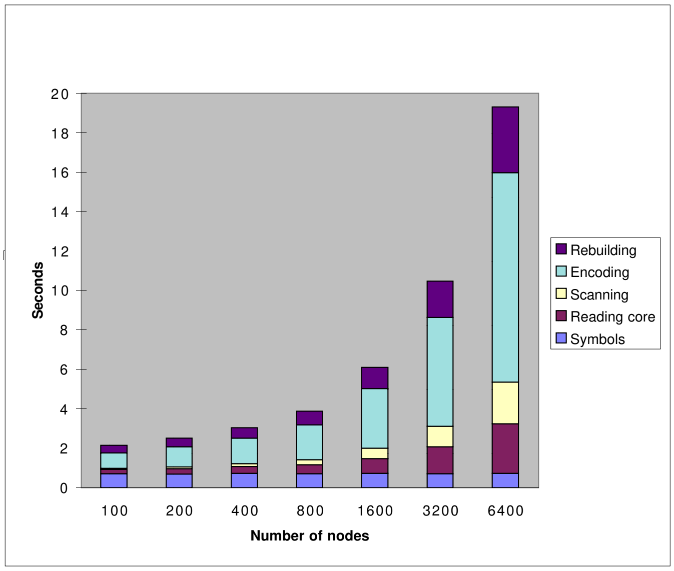

Figure 5: “tree” on i486

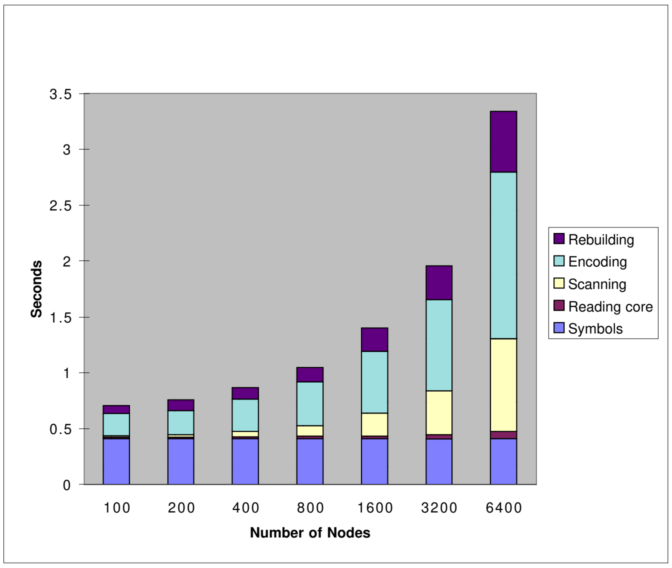

Figure 6: “tree” on PowerPC

of the performance for each of the three programs. To avoid paging problems, all tests were performed on the same large machine.

- `tree` – This program has linear performance for all three components. This is expected for encoding and rebuilding, but is not guaranteed for scanning, since each splay tree insertion could have shown the worst case performance of O($lg\ n$). However, for this program, each insertion is being performed in very close to constant time.

- `fibonacci` – Again, the encoding and rebuilding components are linear, but the scanning cost is O(n^2). This is because `migrout` scans the stack in an approximately linear fashion (in order of addresses). Insertion of new data items into the splay tree is in an increasing order. This results in a splay tree that resembles a linear linked list.

- `arrays` – Since there are only 50 arrays, the scanning component requires an insignificant amount of time to locate them. However, since each array is large, the encoding and rebuilding components are significant, although they will always have O(n) complexity.

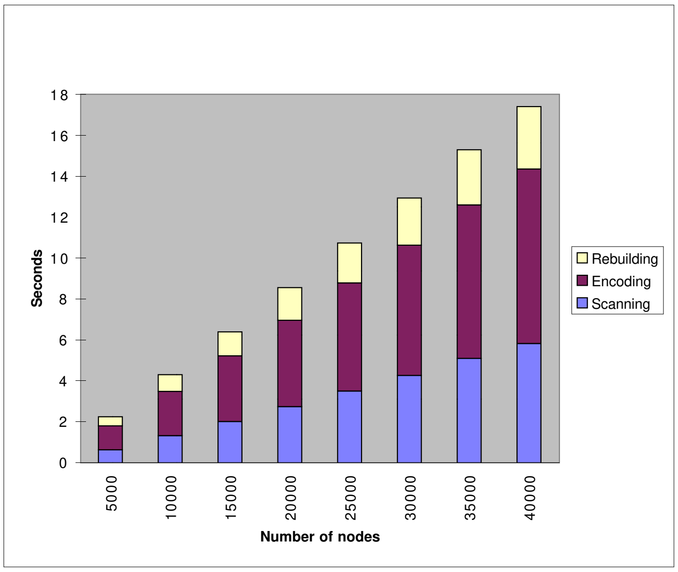

Figure 7: Growth of “tree”

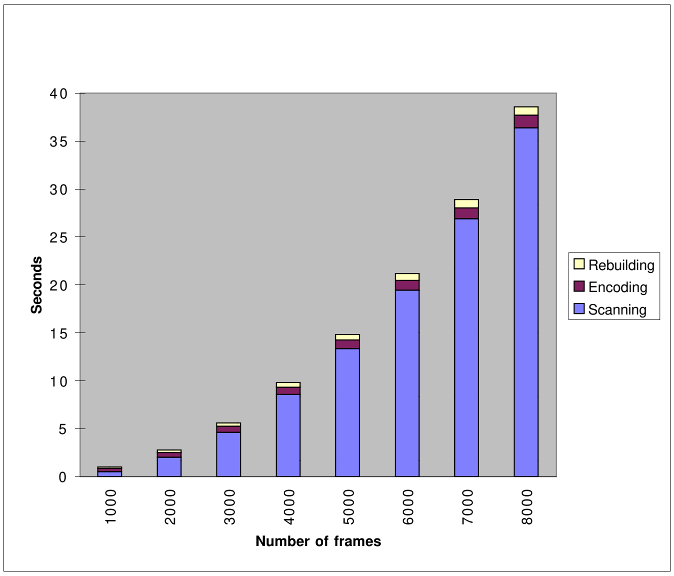

Figure 8: Growth of “fibonacci”

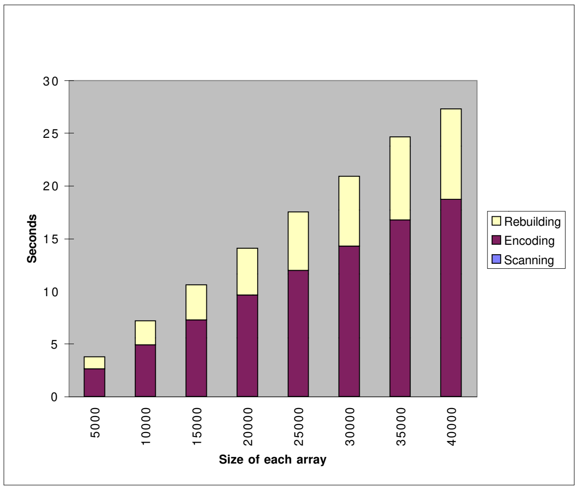

Figure 9: Growth of "arrays"

5.3 Improving Performance

If Tui is to be used in a realistic environment, the total running time must be minimal. From the previous analysis, we determine that small programs can be migrated within 5 seconds, causing a slight disturbance to the user. However, with larger (more realistic) programs, the delay could increase by several orders of magnitude. Even though these delays would be intolerable for interactive use, for long distance and wireless computing the cost of migration may still be insignificant when compared to the cost of executing the program over expensive communication links.

There are several optimizations that should be made to the current prototype version of Tui that will substantially improve performance.

- Reading the debugging symbols and creating the type tree is a fixed cost that can be avoided by preprocessing that data. If the tree can be built at compile time, then both `migrin` and `migrout` only need to read the type tree directly from a file, rather than translating it from the stabs format.

- As seen in the case of the fibonacci program (migration time of O(n^2)), the splay tree is not always the best structure for storing the value table. For random accesses (as in the tree program), performance is almost optimal, but for inserting stack values in sequential order, the performance is at its worst.

  The solution is to use a table into which the global and stack variables are added. This is done in address order, so that all additions to the table are O(1). When scanning the heap space, accesses must be randomly ordered, therefore the table should be converted into a balanced splay tree.

- The whole program would benefit from a more efficient coding style. That is, excessive debugging information could be removed, and the remaining code should be optimized by the programmer.

- Use of faster computers in the future will reduce the total running cost. This cost will become insignificant in comparison to the large cost of executing programs or accessing remote data.

6 Enhancement – Dealing with non-migratible language features

The current implementation of Tui is only a prototype. It has been shown to function correctly when migrating a moderate range of simple programs, but it has not yet been tested on randomly chosen software. The primary advantage of Tui, over the Emerald mobility system, is that it focusses on languages that are not type safe (such as ANSI-C). Tui must either successfully migrate any program, or must inform the user of why this is not possible. Therefore, a extensive survey of ANSI-C programs and language features must be made.

There are several points in time at which a program may be declared to be non-migratible.

- **Compile time** — The compiler will detect a language feature that is guaranteed to make the program non-migratible. A special compiler warning will be given, allowing the programmer to reconsider the use of that feature. It is expected that most problems will be detected at this phase.

- **Run time** — A run time check will notice that an illegal action has taken place. This is not a desirable time to perform checks, due to performance degradation.

- **Migrate time** - If Tui detects a type inconsistency during migration, a detailed report of the clash is given. This will normally only happen if the program contains a type error that was not detected at compile time or run time.

Examples

The following list shows several of the non-migratible features of ANSI-C that have been identified to date.

- `sizeof` — This operator returns the number of bytes of memory storage that a variable of a given type requires. Since this value depends entirely on the host machine, it may become incorrect after migration. One solution is to disallow the use of `sizeof`, since it suggests that the program might be non-portable.

However, it must still remain legal in situations such as allocating memory with `malloc`.

- **Void pointers** — When a pointer value is assigned to a `void *`, we no longer know the type of the data that is being pointed to. The solution is to enlarge each void pointer with a tag that specifies the pointer's base type. Every time an assignment is made to a void pointer, the correct type number must also be stored. Upon retrieving a value, a run time type check ensures type consistency. This solution involves extra code generation for tagging and checking pointers.

- **argv array** — This list of arguments is similar to a normal C array, but since the size varies at run time, it is not possible to correctly specify the type. Extra migration time code will fetch the `argc` variable to complete the type details.

- **Pointer casting** — Casting between non-void pointer types is normally considered non-migratable since we are creating a type inconsistency. That is, the original pointer implies that the addressed memory has one type, whereas the newly created pointer implies a different type. In some situations (such as the `memcpy` procedure), it is necessary to create a `char *` from whatever pointer type is passed in. The procedure can now access the data as a raw sequence of bytes.

  A solution to this problem is to detect the scope of the cast, and to allow casting that only affects the current procedure. That is, if an illegal assignment is made, but the lifetime of the resulting value is contained within a single procedure, then removing preemption points from that procedure will solve the problem. On the other hand, if the pointer is stored in a global variable or on the heap, or is passed to another procedure, then conflicting type information may exist at migration time.

To complete the study of non-migratable features, we intend to survey other languages such as Pascal, Modula-2 and Fortran (ACK already supports these languages). It is expected that the problems and solutions will be similar to those of C, but a few new problems may arise.

7 Related Work

There is very little previous work that is directly related to heterogeneous process migration. As previously stated, only Emerald is known to support the transfer of active programs between different architectures. There are however many areas of research related to heterogeneous migration. This section gives a list of such areas, showing their relationship to heterogeneous migration.

- **Traditional Migration Systems** — Process migration is not a new topic, and has been studied extensively since the late 1970s. Examples of process migration systems are V [9][37], Charlotte [4], DEMOS/MP [27], Sprite [12], Condor [8] and Accent [40]. A good summary of these and other systems is given in [25].

Much of the previous research has involved finding new and improved methods of transferring the state of the process from one machine to another. In all of these systems, the process can only be migrated between homogeneous machines. The Tui system assumes that the underlying operating system is capable of some type of homogeneous migration, and adds the further option of allowing processor heterogeneity.

- **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 [5] [21] [28], DOWL [3], DCE++ [29], and COOL [22]. As previously mentioned, Emerald is the only system that permits heterogeneous mobility of active code.

- **Debugging** — Source level debugging is one of the most closely related topics 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.

  Although debuggers are similar in style to the Tui system, they have some major limitations. They are able to give the user a good representation of the program's data, but they are not always exact. For example, if a C program contains a `union` data type, the debugger will either refuse to show the data, or will show all possible options, requiring the programmer to decide on the correct value. Also, if generic pointers (such as `void *` in C) are used, the debugger will not be able to dereference the pointer unless the user performs explicit type casting.

  Recently, work has progressed in the field of debugging optimized code [11] [19]. 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 [1] is a newly developed format that is capable of expressing the structure of an optimized program.

- **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 [31]. There was a desire to convert existing VAX software to the Alpha platform, without using the original source code. Another system [30] 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.

- **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 [24] and ISO's ASN.1 [2].

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

  One solution [18] 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 by 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 traditional garbage collection techniques is given in [10], and some of the more recent issues are discussed in [17]. See [6] and [38] for some further examples.

  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 [7] 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 a second system [16], the compiler is extended so that an extra garbage collector function is automatically generated for each standard function in the program. When garbage collection takes place, the entire stack is traversed on a frame by frame basis, with the appropriate function being called for each. These functions perform collection by following any pointers that are present in that frame. As with the Emerald migration system, this algorithm will only function correctly in a strongly typed environment.

- **Heterogeneous Distributed Shared Memory** — The *Mermaid* system [39] [41] 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.

- **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 [8] [26], but they only function in a homogeneous environment.

8 Summary

The Tui Heterogeneous Process Migration system is able to move a process between machines of different architecture. It uses type information that is generated when the migratible program is compiled. The bulk of the 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 have shown that migration is possible within a tolerable amount of time, although further optimizations to the data structures and algorithms will be necessary.

Tui has been designed to migrate ANSI-C programs, although other languages such as Pascal and Fortran would be similar. These languages are not type-safe, so locating data is not always possible. Proposed future work will involve categorizing non-migratible language features, and attempting to eradicate them.

References

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

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

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

[4] Yeshayahu Artsy and Ralph Finkel. Designing a Process Migration Facility: The Charlotte Experience. *COMPUTER*, September 1989.

[5] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and Abstract Types in Emerald. *IEEE Transaction on Software Engineering*, January 1987.

[6] Hans J. Boehm, Alan J. Demers, and Scott Shenker. Mostly Parallel Garbage Collection. *ACM SIGPLAN Notices*, June 1991.

[7] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, September 1988.

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

[9] David R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3), 1988.

[10] Jacques Cohen. Garbage Collection of Linked Data Structures. *ACM Computer Surveys*, September 1981.

[11] Max Copperman. Debugging optimized code without being misled. *ACM Transactions on Programming Languages and Systems*, page 387, May 1994.

[12] Fred Douglis. Transparent Process Migration : Design Alternatives and the Sprite Implementation. *Software Practice and Experience*, August 1991.

[13] 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.

[14] 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.

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

[16] Benjamin Goldberg. Tag Free Garbage Collection for Strongly Typed Programming Languages. *ACM SIGPLAN Notices*, June 1991.

[17] Xiaomei Han. Memory Reclamation in Emerald - An Object Oriented Programming Language. Master's thesis, University of British Columbia, 1994.

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

[19] 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.

[20] Wilso C. Hsieh, Paul Wang, and William E. Weihl. Computation Migration: Enhancing Locality for Distributed Memory Parallel Systems. *SIGPLAN Notices*, page 239, July 1993.

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

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

[23] Julia Menapace, Jim Kingdon, and David MacKenzie. The "stabs" debug format. Technical report, Cygnus support.

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

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

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

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

[28] 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*, January 1991.

[29] 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.

[30] Gabriel M. Silberman and Kemal Ebcioglu. An Architectural Framework for Supporting Heterogeneous Instuction-Set Architectures. *IEEE Computer*, June 1993.

[31] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary Translation. *Communications of the ACM*, February 1993.

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

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

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

[35] 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.

[36] A.S. Tanenbaum, H. van Staveren, E.G. Keizer, and J.W. Stevenson. A UNIX Toolkit for Making Portable Compilers. *Communications of the ACM*, September 1983.

[37] 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.

[38] Ian Toyn and Alan J. Dix. Efficient Binary Transfer of Pointer Structures. *Software - Practice and Experience*, November 1994.

[39] D. B. Wortman, S. Zhou, and S. Fink. Automating Data Conversion for Heterogeneous Distributed Shared Memory. *Software Practice and Experience*, page 111, September 1994.

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

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