Abstract

Heterogeneous process migration involves moving a process between machines that have differing hardware and software configurations, including different processor architectures, machine instruction sets, and operating systems. This paper describes an approach to heterogeneous process migration that involves building a machine-independent "migration program" that specifies the current code and data state of the process to be migrated. When this program is compiled and executed on the target machine, it will first reconstruct the process's state and then continue the normal execution of the now-migrated process. The principal advantage of this approach is that it hides the details of code and data translation in the compilers for each machine.

1 Introduction

In a distributed system supporting transparent remote execution of applications, process migration can be useful for a variety of reasons:

- It can be used to reduce the disruption of activities such as maintenance and repair of hardware.
- It can be used to improve the performance of individual processes and an entire system in the face of changing resource availabilities. For example, Leland and Ott [2] have shown that many systems can be characterized as containing many small jobs and a few "hogs" that consume the vast majority of all computing cycles. These hog processes typically run for extended periods of time and considerable efforts can be justified to ensure that they are properly spread among the available computing nodes.
- It provides a second axis of optimization for a "data browser" class of applications that interact with various large databases, reading and writing data from one database for a while, and then moving on to another. Optimal behavior for these applications involves trading off both data motion and program motion.

Process migration has already received extensive study for the homogeneous case, where both the source and target machine of a migrating process consist of the same kind of hardware and run the same underlying systems software. Homogeneous process migration can be done without knowing all the details of the process state being migrated because the code and data elements involved can be copied without being understood.

Heterogeneous process migration involves moving a process between machines that have differing hardware and software configurations. To migrate a process between heterogeneous bases requires sufficient knowledge about a process's state to be able to create an equivalent state on the target machine, so that the migration is transparent to the rest of the world. A prerequisite concern is that the state of a process be sufficiently specified to make such a translation possible.

This paper concerns itself primarily with those issues of heterogeneous process migration that are not already addressed by the requirements of homogeneous process migration. We are concerned with the issue of translating the running state of a process from its representation on one machine to an equivalent representation on another machine. We propose to implement migration by means of recompilation, building a machine-independent migration program that specifies the state of the process to be migrated.

When this program is compiled and executed on the target machine, it will first reconstruct all of the global and heap data from the process on the source machine and then execute one chain of calls for each stack to modified versions of the procedures that were on the
call stacks of the process on the source machine. These modified procedures will first reconstruct the stack data and then resume execution of the now migrated process.

Many programming languages do not specify the semantics of their behavior exactly or in a machine-independent manner. For example, the same floating point operation may yield different results on different machines. Machine-dependent values like the size of data types may be available as part of the language, allowing programs to exhibit machine-dependent behavior. If the behavior of a program depends on the underlying hardware, it is senseless to ask the semantics of transparent migration.

Assuming that the incarnation of a program as a process on one machine can be translated to an equivalent incarnation of the program as a process on another machine, there is still the question of how to actually implement this translation. In general, it will require detailed knowledge of the hardware and systems software of each machine involved. This knowledge already exists in the compilers and debuggers for each machine. We would like to be able to take advantage of that knowledge and avoid recreating it. By specifying the state of a process in terms of a program and compiling that program we achieve this goal. However, specifying a process's state in terms of a high-level programming language is difficult for some languages. If an intermediate language is available, such as one used to communicate between compiler front-ends and back-end code generators, then process state specification may be considerably simplified.

Finally, efficiency is always a concern. The migration program for a process always includes the program that the process is executing. Full recompilation would add substantially to the overhead of migration, but partial recompilation techniques may be employed to reduce this overhead to a more acceptable level.

The rest of this paper addresses these issues in more detail. Section 2 summarizes other approaches to the topic. Section 3 describes our model of computation and the requirements placed on programming languages for which our approach will work. Section 4 presents the approach we propose for heterogeneous process migration, while section 5 examines the costs of our approach. Section 6 finishes with conclusions and suggestions for future work.

2 Related Work

Homogeneous process migration is a fairly well understood operation [5, 8, 9, 4]. When the two hardware platforms are not the same, solutions to date work only in special cases [6] or by interpretation of the source code of a program [1]. The latter approach is in reality a case of homogeneous process migration since the machine—the interpreter—is the same on all hardware. The disadvantage of this approach is that it sacrifices the performance obtainable from running programs that have been compiled to the native instruction set of a physical machine.

We are aware of one effort, by Shub [6], based on the former approach. This effort has focused on a restricted class of C programs and has not specified how translation between source and target program states could be performed in a general, automated fashion. C programs are not allowed to alias pointers, so that the state of a program's data can be determined from its binary object file and from special tables kept by a modified heap storage allocation package. This prohibits all C pointer coercions, including the common case of "narrowing" a generic C pointer to be a pointer to a specific data type. Compiler optimizations involving code motion are also disallowed.

The translation techniques used by Shub are based on a combination of "hand-coded" translation procedures and the availability of a single compiler that can target every machine existing in a system. A common compiler allows for space allocation policies that simplify data element translation. For example, structure packing can be guaranteed to be the same on all machines and the storage size for a data type can be made to be the maximum required by any machine. Identical optimizations are compiled for a program without regard for target machine, so that the states of the process on one machine correspond to the states on any other. The compiler also directs all calls to heap storage management to a modified package that generates runtime symbol tables describing the data layout of the heap. The actual translation of process state is done by means of hand-coded migration procedures that use the symbol tables generated by the compiler and the heap storage management package to find all the data elements of a process. Note that these translation procedures must be coded for every machine type existing in a system and are compiler dependent.

For systems with failure recovery, process migration can be implemented in an entirely different manner. The process to migrate is destroyed on the source machine and then recovered on the target machine. If the information logged for process recovery is written in a machine-independent manner then this approach can handle the heterogeneous process migration case.

We know of no system that has actually implemented such an approach, but Argus [3] supplies the necessary facilities to do so in theory. The tradeoff of this
approach compared to others—including ours—is that failure recovery facilities typically impose an overhead throughout the execution of a program. State checkpoints are periodically written out to stable storage. Our approach imposes an overhead only at migration time and hence should be more desirable for migratory processes that don’t need failure recovery capabilities.

Our approach is similar to that of Shub in that we deal with compiled binary programs rather than interpreted source programs. However, we avoid the details of machine and compiler-dependent data translation procedures by employing recompilation techniques to keep knowledge of these matters hidden within the compilers and debuggers. Also, by going to this level of abstraction, our approach extends easily to a large class of programming languages rather than being specifically targeted for the implementation idiosyncrasies of a single programming language and compiler. The price we pay for this is an increase in the time to migrate, since migration now involves recompilation and relinking of various parts of the program instead of direct manipulation of memory values at absolute addresses in a program binary image.

3 Model of Computation

3.1 Abstract and physical program states

The key notion of our approach is that a programming language defines an abstract machine on which a program can be run. At many points in the program’s execution on physical devices its state can be specified in terms of the current state of this abstract machine. Intuitively, these are the points in the execution of a program where a source-level debugger could make sense of the state. We migrate a process by automatically writing a program that first reconstructs the data, and then continues on with the processing where the source process left off. There are many ways to reconstruct the data, and we will compare some of their efficiencies later.

Some programming languages have the ability to “undo” their execution to a previous state. Two examples of such languages are Prolog and Snobol. The techniques for migrating programs in such languages are substantially different, since execution history is accessible to the program. We exclude languages with such backtracking facilities, and focus only on languages with easily reconstructible state.

Compilers translate a source program for an abstract machine into a binary program for a physical machine that produces the same external behavior as the source program. “Behavior” means externally visible state, such as the values of data elements as they are defined and understood by the world surrounding a program, at the points where the program interacts with the world.

The compiler is free to change the program in any way that does not change its external behavior. Compilers strive to find ways to cut resource use, while maintaining behavior. With an optimizing compiler, the internal states of the source program on the abstract machine and the binary program on the physical machine will correspond only at a subset of the execution points of each program.

At an execution point where the states of the abstract and physical machines correspond we can represent the state of the binary machine program by means of a source program that describes the corresponding abstract machine state. We call such points migration points. We can invoke the same primitives used for source-level debugging to reverse-compile the machine-dependent binary program state to a machine-independent source program description.

3.2 Code Optimization

3.2.1 Frequency of Migration Points

To keep the delays in migration small, there should be as many migration points in a program as possible, so that the option to migrate is available as frequently as possible. If the compiler used does not perform any optimizations then the number of migration points available will be maximized. Unfortunately, inhibiting optimizations also imposes a heavy penalty on the program’s execution time.

What we really want is a bound on the time required for a program to reach a migration point from the time that migration is requested. Restating our requirement in this fashion allows any optimizations to occur, including rearranging basic code blocks and optimizing loops, as long as we can guarantee such behavior. Even complex optimizations are allowed if fix-up code invoked at migration brings the physical state in line with an abstract state.

If we assume time bounds for migration on the order of a second, then this leaves room for several million machine instructions to occur before a migration point must be reached. Migration points might have to be inserted into long blocks and loops to meet the requirements.

If a procedure on the stack, but not currently executing, is not in a state corresponding to an abstract state, we cannot migrate easily. We must wait for the procedure it has called to return, and then let it run to
a migration point. Some procedures may always execute in a short time, and we can wait for these calls to terminate. For calls to procedures with no such guarantee, we require that the call point also be a migration point, and so at migration time any procedure on a call stack will be at a migration point.

Migration points are also abundant if a program periodically returns to a predetermined migration point. To migrate individual objects in an object-oriented system we can make the "waiting for message" state of each object a migration point. If we can assure that every method invocation terminates within the time bounds we require for reaching a migration point, then the compiler is free to optimize the individual methods without restrictions.

### 3.2.2 Compiler Intermediate Languages

The abstract language used for the migration program could be a high-level language, but it is more convenient to use a lower-level language such as the language used to communicate between a language-dependent compiler front-end, and a machine-dependent code generator. The machine-specific optimizations available in the code generator are then available to the migrating process, but we can avoid much of the expense of language processing. This approach does not even require all machines to share a retargetable compiler, but each pair of machines must understand a common language, and the migration program must be written in the appropriate language.

### 3.3 Completeness of the Language Specification

There are several important requirements we make of the abstract machine specification for a programming language in order to support our approach to process migration. These center around the need to specify the current state of a program running on the abstract machine in sufficient detail to be able to translate between it and some equivalent physical machine state. There can be no ambiguity about the meaning of a program's data elements or the operations that may be performed on them. Unfortunately, this is not the case for most programming languages. For example, the specification of floating point numbers and the results of various operations on them is frequently ill-specified. Consequently the results of performing the same operations on the same data on two different machines may differ because the compiler used for each machine is free to implement the ill- or unspecified aspects of the semantics in any fashion it chooses.

I/O operations are also frequently ill-specified. Many programming languages simply do not specify how activities such as I/O are performed. Others specify behavior in terms of a standard "system procedure library". We must wait for any such operations to terminate before we migrate, and rely on the network-transparent nature of the execution environment to allow the process to migrate after the call.

We must also disallow machine-specific variables, such as the storage sizes of data types, and exclude from our consideration programs whose behavior can depend on machine-dependent values. Many programs are portable in the sense that they can be started and successfully run on a given machine independent of its hardware configuration, but these programs are not machine-independent—and hence not migratable—since their behavior will differ from machine to machine. Their source specification is in fact the specification for a class of programs. An illustrative example of a portable, machine-dependent program is a program that packages data records into a fixed size byte array buffer to ship across a network. The number of records that will fit into the buffer will depend on the size of the records on the specific physical machine.

Programs are not allowed to have "hidden knowledge" of their abstract state. A program with untagged union data, for example, may be able to remember which branch of the union is appropriate and never access the data through an inappropriate branch, but this knowledge cannot generally be discovered. It is generally difficult to prove that such programs have no type errors, and discovering the data type in the absence of tagged data or tagged pointers is equally difficult.

In general, type-safe languages are more likely to meet our requirements. Even these sometimes do a poor job of specifying how floating point operations or I/O behave. Such languages limit the manufacture and type coercion of data references and hence prevent the hiding of information.

### 3.4 Problem Definition

At this point we are ready to specify what we mean by the term *process migration*. Our goal is to suspend a running process on one machine, copy its state to a newly created process on a second machine of possibly differing hardware and software configuration, destroy the first process, and set the second process running in a fashion such that the rest of the world remains unaware of the change unless an explicit query for the location of the logical process is made.

This goal requires all the preconditions needed for the well-studied topic of homogeneous process migra-
tion, such as network-transparent execution environments and network-transparent IPC [7]. We will not address these aspects of the problem here, other than to assume their solution. We will focus only on the question of copying and translating the state of a process on the source machine to an equivalent process state on the target machine.

We assume that the programs being run by processes contain migration points and that there exists a means of suspending a process at a migration point. This might require, for example, suspending a process, setting a breakpoint at some migration point, and then resuming the process. We assume that the compilers ensure that migration points occur frequently enough to satisfy the time bounds on migration for a system.

Finally, we assume that programs are written in a programming language whose state at any migration point is sufficiently well-specified to allow its complete translation between machine-dependent and machine-independent forms.

4 Migration Through Recompilation

4.1 The Machine-Independent Migration Program

The outline of the approach we propose for process migration can be described as follows:

- Suspend the process to migrate on the source machine.
- Translate the machine dependent state of the process into a machine independent state.
- Create a machine-independent "migration program" that represents this state.
- Compile, link, and load the machine-independent program on the target machine.
- Destroy the source process.
- Run the loaded program on the target machine to recreate the migrated process.

We assume the problems of maintaining consistent external connections, such as IPC, in the face of migration have already been solved, and the only remaining problem is to create the migration program.

The machine-independent migration program must first recreate the current data state of the process to migrate and then return control to the appropriate point of that program. We will use modified versions of each procedure on the stack to first recreate the stack data, and then resume the execution of the original procedure at an appropriate point.

There are typically three kinds of data space in a program: global data, heap data, and procedure local data. The global and heap data can be recreated by a global initialization procedure that can be run at the beginning of the migration program, before any of the modified stack procedures are called.

In a language where jumps may be made to any point in a procedure, we can construct a modified call stack procedure in a simple and separable manner. Consider a call stack where procedure A calls procedure B, which in turn calls procedure C. The modified version of B, call it B1, will consist of the following parts:

1. Code to initialize all local variables.
2. A call to the modified procedure C1.
3. A jump to the point just past the call to C in a copy of the code from the original procedure B.
4. A copy of the code of B.

More specifically, consider the following pseudo code version of B:

```
procedure B:
    s0;
    for (i=0; i<n; i++) {
        s1;
        c();
        s2;
    }  
    s3;
```

If the loop has been executed fully four times, the value of i is now 4, and we are calling C for the fifth time. The transformation to make B1 would be:

```
procedure B1:
    InitLocalVars;
    /* This includes i := 4 */
    c1();
    goto L;
    /* Copy of B, with label */
    s0;
    for (i=0; i<n; i++) {
        s1;
        c();
    }  
    L:
        s2;
    }  
    s3;
```
This procedure initializes its local variables, and calls other modified routines to initialize the variables of frames further up the stack. When C1 returns, it will have done all the work the second invocation of C was committed to before the migration, and the goto will cause resumption of the work B was committed to. If the code represented by s0 is now dead, the compiler is free to eliminate it. Note that the next time around the loop C, not C1, will be called. Each modified procedure is called only once.

To appreciate the advantage of not using a high-level language for the migration program, consider what would happen if the language disallowed the goto into the loop in the above example. The portions of the loop not yet executed must be completed before the goto is performed, and the target of the goto is moved past the end of the loop.

```plaintext
procedure B1:
  InitLocalVars;
  /* This includes i := 4 */
  C1();
  s2;
  for (i=5; i<n; i++) {
    s1;
    C();
    s2;
  }
  goto L;
  /* Copy of B with label */
  s0;
  for (i=0; i<n; i++) {
    s1;
    C();
    s2;
  }
L:
  s3;
```

This rewriting can be arbitrarily complex and lead to code explosion, depending on the restrictions in the language. The code above could be more complicated if s1 or s2 contained break or continue statements, gotos, labels or assignments to the loop variable. The complexities involved in rewriting for a more restrictive high-level language argue for using a less restrictive intermediate language. However, our approach will work for high-level languages, including those similar to Modula-2, type-safe Cedar, and Lisp.

4.2 Translating and Sending the State of a Process

To construct our migration program we need to translate the entire current state of a process from its machine dependent form to a machine independent form. We assume that the compilers generate source-level symbol tables describing the locations at which one can find any global and procedure-local variable. The global and procedure-local state can be evaluated in the same fashion that a source-level debugger can be asked to list the state of any given procedure on a call stack or of all global variables. A procedural interface to such a source-level debugger would be sufficient to gather the information we need.

To find the state of the heap, we must trace it in a fashion similar to that used by tracing garbage collectors. We must follow each pointer variable in the global and call stack state to find the transitive closure of the data elements they point to. Again, we assume that the abstract program state is sufficiently self-describing that we can correctly interpret all pointers encountered. Note that in addition to interpreting the pointers, which is all that garbage collection requires, we must also be able to interpret every field of every heap object.

Once we have the state of a process, we must still send it to the target machine. Since the language of the migration program is determined by the available compilers, it may be inefficient to transmit the migration program between machines in this language. In addition, the migration program is highly idiomatic, containing many elements built from a few simple templates. We can reduce the communication cost associated with migration by employing compaction techniques to obtain a shorter representation of the migration program.

There are even better ways to handle some idiomatic constructions. For example, the initialization of an array in the migration program can be compact, but still explicitly includes each data value in the array, and we would like to avoid making the compiler process these constants. Instead, we may write the array to a file and include in the migration program code to read the file. If the migration program is compiled and run before the source process is destroyed, we can avoid explicitly creating this file and use a network connection to stream the data directly to the running migration program.

4.3 Recompilation

At the conceptual level, we simply recompile a migration program in its entirety for the target machine. If we assume the preexistence of a binary program ver-
occurred. With the correct choice of intermediate language, each new procedure is just like its doppelganger, except for the addition of a preamble, and only these preambles need to be compiled.

In addition, we should be able to take advantage of incremental linking. The new procedures are called at migration and never from the original code, so the linker only needs to deal with the new procedures and doesn't need to change or add any references in the previously linked code.

If the migration program is specified in an intermediate language then several additional efficiencies should be obtainable. To begin with, recompilation of the various migration procedures will be significantly cheaper if none of the language-specific steps are performed and the machine-independent optimizations have already occurred.

5 Performance Issues

The dominant cost for homogeneous process migration is the cost of copying state between machines. For heterogeneous process migration there are additional costs of comparable size for translation of the data state of a process to the machine-independent form and for recompilation and relinking the migration program.

These three activities can be overlapped. If data are streamed directly from the source machine to the newly created process on the target machine then the extraction, transmission, and insertion of this state can be overlapped. Depending on the hardware parallelism available in the two machines, this overlapping could reduce the cost of migrating this part of the process's state by a factor of 2 or 3.

Since the binary of the original program must be linked with the modified procedures, we shouldn't begin a migration until the binary is present on the target machine. We can continue executing the process on the source machine until the binary is recreated on the target machine or fetched from some central server.

Employing such techniques to reduce our migration time cost is especially important because our approach to state specification prevents any of the VM paging techniques used in homogeneous process migration, such as precopying state or demand-paging state, to reduce the time during which a migrating process is suspended. Fortunately, by "precopying" the target binary, overlapping the processing of the global and heap data, and employing incremental recompilation techniques, we may be able to keep the cost of migration relatively low.

6 Summary and Conclusions

For programs written in a programming language with sufficient semantic clarity we have described a method by which they can be migrated between heterogeneous hardware and software bases. We believe our method should work for several important classes of languages, including those characterized by languages such as Modula-2, type-safe Cedar, and Lisp.

The approach we have taken emphasizes hardware and software independence at the expense of migration time cost. By employing recompilation and source-level debuggers we hide the details of migration within the compilers and debuggers for a language for each machine. This implies that the details of data and code translation need not be addressed by the implementor of migration.

Equally importantly, our approach allows us to migrate between different operating system bases since we rely on the abstract definition of a language's system library interface. Any operating system that correctly implements the semantics of the programming language used will be transparent to our migration activities.

Finally, the approach of recompilation is not tailored to any particular programming language. However, using a compiler intermediate language for migration has several significant benefits. Most important of these is that frequent migration points can be generated in source programs without hindering compiler optimizations. This allows a migration capability that imposes overhead on programs only at the point of migration instead of on the entire execution cost of every program run.

Another interesting capability afforded by the use of a compiler intermediate language is that we can migrate mixed-language programs, and automatically gain migration for programs in a new language when its front end is completed.

Our method entails additional time cost for migration. Migration now includes creation and compilation of state initialization code, in addition to the cost of sending a process's state between machines. Incremental recompilation and relinking techniques help mitigate these costs.

Hardware independence implies that we cannot employ VM paging techniques to precopy or demand-page process state between machines. We must suspend the migrating process for the entire time needed to migrate its state, which is measured in seconds, rather than suspending it for only a brief, subsecond "initialization"
period and letting it run in parallel with the migration activity for most of the time.

There is still much work to be done to understand how our approach to heterogeneous process migration would work in practice. Foremost is the need for an implementation and performance evaluation to determine the actual overheads. Despite the additional costs of heterogeneous process migration over homogeneous migration, we feel that many applications will benefit.

Most programming languages do not specify their entire semantics, falling down when it comes to such topics as floating point numbers and I/O operations. We can approach this wall from both sides. Pragmatically, we can explore exactly how far from the formal requirements a real programming language can get and still allow reasonably coherent migration. For example, while different hardware floating-point implementations cause problems a "best effort" at migration would produce reasonable results for many programs. At the same time, it would also be interesting to determine what subsets of some popular languages, such as C or C++, would be suitable for our approach. For example, one could envision a preprocessor for appropriate C and C++ subsets that could be employed to "certify" that a program is suitable for migration.

References


