# Object and Native Code Thread Mobility Among Heterogeneous Computers

Bjarne Steensgaard*
Microsoft Research
One Microsoft Way
Redmond, WA 98052
USA

Eric Jul
DIKU (Dept. of Computer Science)
University of Copenhagen
Universitetsparken 1
DK-2100 København Ø
Denmark

### Abstract

We present a technique for moving objects and threads among heterogeneous computers at the native code level. To enable mobility of threads running native code, we convert thread states among machine-dependent and machine-independent formats. We introduce the concept of *bus stops*, which are machine-independent representations of program points as represented by program counter values. The concept of bus stops can be used also for other purposes, *e.g.*, to aid inspecting and debugging optimized code, garbage collection etc. We also discuss techniques for thread mobility among processors executing differently optimized codes.

We demonstrate the viability of our ideas by providing a prototype implementation of object and thread mobility among heterogeneous computers. The prototype uses the Emerald distributed programming language without modification; we have merely extended the Emerald runtime system and the code generator of the Emerald compiler. Our extensions allow object and thread mobility among VAX, Sun-3, HP9000/300, and Sun SPARC workstations. The excellent intra-node performance of the original homogeneous Emerald is retained: migrated threads run at native code speed before and after migration; the same speed as on homogeneous Emerald and close to C code performance. Our implementation of mobility has not been optimized: thread mobility and trans-architecture invocations take about 60% longer than in the homogeneous implementation.

We believe this is the first implementation of full object and thread mobility among heterogeneous computers with threads executing native code.

## 1 Introduction

A trend in distributed operating systems has been to either support communication and remote procedure call [BN84] among heterogeneous computers [BCL+87, Gib87] or to support object and thread/process mobility among homogeneous computers [Jul89, Dou87]. We have combined the two by extending the Emerald system [BHJL86, BHJ+87, Hut87, HRB+87, JLHB88, Jul89, RTL+91] to support object and native code thread mobility among heterogeneous computers connected in a local network as shown in Figure 1.

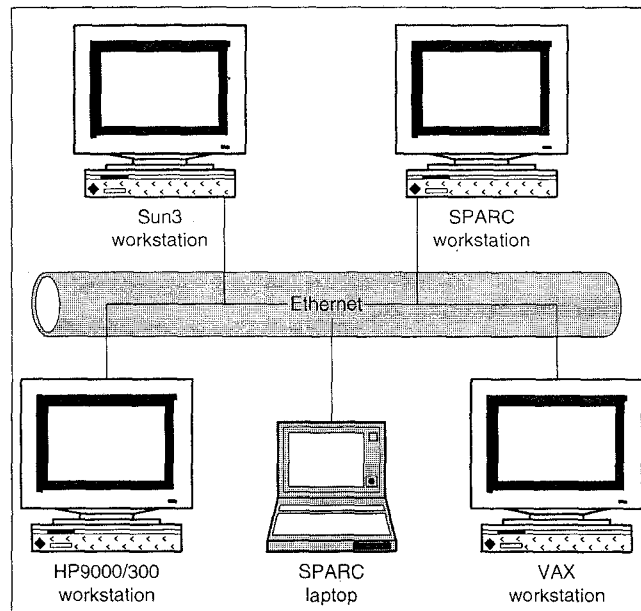


Figure 1: A network of heterogeneous nodes. Sample configuration of a local network with heterogeneous workstations among which we are able to move both objects and native code threads in our prototype implementation.

In this paper, we describe the problems encountered when enhancing a homogeneous object system with mobility to support heterogeneous architectures. We present the concrete techniques used in our implementation and explain how these techniques are special cases of more general methods for mapping program states among machine-dependent and machine-independent formats. We provide performance numbers for homogeneous and heterogeneous thread migration between four different architectures.

By *object mobility* we mean that an object in an object based programming system is able to physically change location within a set of processor nodes (in our case: workstations). Mobility is fine-grained in the sense that individual objects, regardless of size, can move independent of other objects residing on the same processor node. Mobility is not restricted to mobility of entire address spaces as in, *e.g.*, the Sprite operating system [Dou87] and the DEMOS/MP operating system [PM83].

By a *thread* we mean a light-weight thread of control running

* Work done while at DIKU, University of Copenhagen, Denmark.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

SIGOPS '95 12/95 CO, USA
© 1995 ACM 0-89791-715-4/95/0012...$3.50

(pseudo-) concurrently with other threads within a single address space.

By *thread mobility* in an object system we mean that a thread is able to move among processor nodes. In the absence of object mobility, thread mobility is nothing but the ability to perform remote procedure calls. In the presence of object mobility, an active thread may be executing an operation in an object that is being moved. When an active thread is "inside" a moving object, the part of the thread state (activation/call stack) describing the thread inside the object must be moved with the object.

**Example 1** Consider an object $X$ residing on node $A$ invoking a operation in an object $Y$ residing on node $B$, the effect of the operation being that $X$ is moved to node $C$. A remote procedure call is performed to invoke the operation in $Y$. When the thread returns from executing the operation in $Y$, execution has to resume on node $C$ where $X$ is now residing. The system has to move part of the call stack of the existing thread from node $A$ to node $C$. □

In our model of thread and object mobility, threads follow objects around as the objects are moved. The reason for this is that Emerald was designed for robust distributed computing: node crashes are considered normal, expected events. We want to minimize residual dependencies [PM83], *e.g.*, by co-locating threads with the objects within which they are executing. Our model differs from the model used in *Oblique* [Car95] where the objects are moved to where the threads are executing.

Object and thread mobility among heterogeneous computers is straightforward, if a system executes machine-independent byte codes and operates on machine-independent data. However, the price of this painless migration is execution inefficiency due to interpretation. Our goal, which we have achieved, is to offer object and thread mobility while retaining the local efficiency of programs that comes from executing native code and operating on machine-dependent data. Thus a thread should run no slower after migration than before and no slower than a comparable thread in a comparable homogeneous system. Furthermore, we provide heterogeneous mobility without *any* language modification.

Object and native code thread mobility among heterogeneous computers is non-trivial because code on heterogeneous computers may differ in the use of registers, number and type of available registers, temporary values in registers or on the stack, instruction sets, program counter values,, data formats, and different levels of optimizations.

The problems caused by differences in use of registers, number and type of registers, temporary values in registers or on the stack, and data formats can all be solved by meticulously keeping track of where different values are placed in object data areas and activation record data areas on different platforms. Such meticulous tracking requires extensive compiler and runtime support. However, this tracking is fundamentally no different than that which is required for supporting homogeneous object and thread mobility. Even in the homogeneous case, the compiler must produce extensive information concerning the location and type of variables that must be converted during the move [Hut87, Jul89]. The advantage of such extensive compiler support is that node-local operations are very efficient—as efficient as comparable C programs—because the runtime overhead is restricted to actual migration operations while non-migration operations are not affected at all.

The problems caused by differences in instruction sets, program counter values, and levels of optimizations are non-trivial because there is no immediate way of translating from code on one architecture to code on another architecture. Two important observations point at a possible solution: 1) object and thread mobility is trivial if we execute machine-independent code and work on machine-independent data, and 2) when moving ordinary data such as numbers and strings among heterogeneous computers we can convert the data to a machine-independent format at the originating node and then translate from the machine-independent format to the machine-dependent format at the receiving node. If we can convert the native code and corresponding program counter values from the machine-dependent format used on a given architecture to a machine-independent format and vice versa then object and thread mobility among heterogeneous computers becomes possible. We introduce the concept of *bus stops* to represent program counter values in a machine-independent manner.

To demonstrate our ideas we have taken an existing object based system with migration, the Emerald system (see [JLHB88][1]), and enhanced it with support for heterogeneous migration. The Emerald language includes constructs for specifying object mobility (and thereby also thread mobility). The language can be used without modification. Our enhanced prototype supports object and native code thread mobility among VAX[2], Sun-3, HP9000/300, and Sun SPARC workstations. The implementation is meant to demonstrate the viability of the concept of object and native code mobility among heterogeneous computers; we have made no attempt to optimize inter-node performance. However, while providing native code mobility we retain the performance advantage of executing native code; intra-node performance on a given processor is independent of whether the thread was created on the processor or migrated to the processor, and is the same as on the original Emerald system, which supports only homogeneous mobility.

We have not attempted to further justify the need for heterogeneous mobility; it should be obvious that any homogeneous migration system can take advantage of transparently becoming a heterogeneous migration system.

The rest of this paper consists of three parts. In Section 2 we discuss general issues related to object and thread mobility among heterogeneous processors. In Section 3 we describe our prototype implementation and present performance numbers. In Section 4 we suggest future work on mobility among heterogeneous processors. Finally, in Sections 5 and 6 we discuss related work and present our conclusions.

## 2 Mobility Issues for Heterogeneous Systems

When discussing mobility of data and threads among processors, it is important to specify the characteristics of the data and the threads to be moved. It is, for example, easy to implement data and thread mobility among heterogeneous processors, if both data and thread states always are represented in machine-independent format. Mobility is much harder, if data or thread states are represented in a format tailored to a specific processor (*e.g.*, native code) as is the case for most efficient systems. In this section, we describe important characteristics of machine-dependent data and thread states and in general discuss how to implement mobility given a certain set of characteristics.

### 2.1 Migrating Data

For performance reasons, most systems use machine-dependent formats for ordinary data such as numbers, structures, strings, and vectors. For example, most systems use the processor's native representation for integers (little or big endian) and floating point numbers (IEEE or non-IEEE). A notable exception is Tcl [Ous94], which represents all types of data as strings.

If data is represented in different machine-dependent formats on two different processors, mobility of data among the processors is typically done by converting the data representation to and from a

[1] Originally presented at SOSP'87

[2] Unfortunately, our last VAX died during this project so our performance numbers are incomplete for the VAX case

commonly agreed upon format. For example, in UNIX implementations, it is common to convert 16 and 32 bit integers to network byte order before sending them over the network (*e.g.*, when performing a remote procedure call) and converting them back to host byte order at the receiving end, *e.g.*, using the `htons(3)`, `ntohs(3)`, `htonl(3)`, and `ntohl(3)` library functions [Sun88]. Heterogeneous RPC implementations usually also support conversion of more complicated data types [BCL+87, Gib87, Sun84, IOS94].

It is also possible for each processor type to use its own machine-dependent format and then convert data between machine-dependent formats as required by each data transfer [SC88]. Unfortunately, the number of conversion routines required is quadratic in the number of data formats. Furthermore, supporting a new data format requires modifying existing systems by adding the necessary data conversion routines.

## 2.2 Migrating Threads

Moving a thread really amounts to moving a thread state. The thread state is essentially composed of a data component representing the values of local variables in the activation records on the call stack and a code dependent component consisting of the thread's executable code and pointers into this code (program counter values). Note, that in an object thread system, file descriptors and similar operating system data is usually represented merely as references to other objects and so is not part of the thread state. The interesting part is the code dependent component because the data component of an activation recored is really no different from normal data which can be moved as described in the previous section.

There are two important characteristics of the code component. The first characteristic concerns the executable code: is it machine-dependent or is it machine-independent? Native machine code is a typical example of machine-dependent code, while source code and byte codes are typical examples of machine-independent code. The second important characteristic is whether or not the code at the originating processor has been subjected to the same transformations and optimizations as the code on the destination processor.

### 2.2.1 Migrating Machine-dependent Code using Bus Stops

In this section, we discuss the problem of migrating machine-dependent code and present the concept of *bus stops* as an implementation technique.

If the code for a migrating object is machine-independent, *e.g.*, byte code, the same code can be executed at both the originating node and the destination node. The issues related to mobility among heterogeneous processors is then no different than mobility among homogeneous processors and can be performed as described in [JLHB88].

However, if the code is machine-dependent, *e.g.*, native code, we cannot execute the same code on heterogeneous processors unless we implement interpreters of the various other machine-dependent formats on each type of processor, which typically is very inefficient. We must therefore have different versions of the code for execution on different types of processors. For example, if we have a thread running on a VAX processor and want to move it to a SPARC processor, we need machine specific versions of the code for both types of processors. How we obtain these machine specific versions is irrelevant in this context.

We see three levels of thread states as illustrated in Figure 2. The top level consists of thread states resulting from interpretation of source code. The middle level represents lower-level machine-independent thread states resulting from execution of, for example, byte code representations of programs. The bottom level represents machine-dependent thread states resulting from execution of, for example, native code. Program execution lower in the hierarchy is typically faster than program execution higher up.

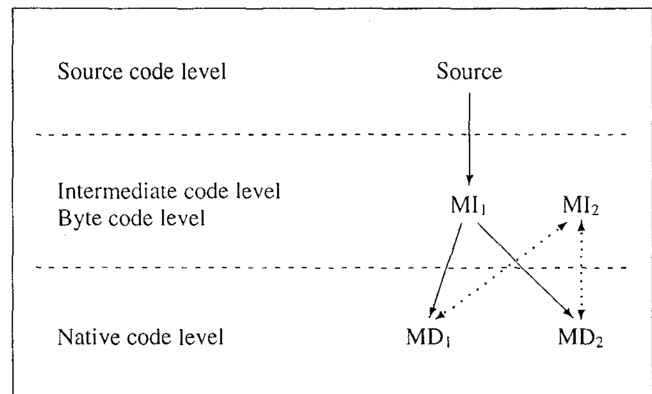


Figure 2: The thread state specialization hierarchy. The "MI" forms are machine-independent forms, and the "MD" forms are machine-dependent forms. The solid arrows illustrate how compilation can statically specialize thread states. The dotted arrows indicate our dynamic transformations of thread states.

The solid arrows illustrate how compiling can specialize program code for efficiency purposes. This transformation is performed statically before the program is executed. The dotted arrows indicate how we implement thread state mobility by transforming a machine-dependent thread state to a machine-independent thread state and specializing the result to a different but semantically identical machine-dependent thread state. Such thread state transformations are performed during program execution when threads are moved.

If the machine-dependent code is native machine code, likely differences between the codes include: non-isomorphic sets of registers, different use of registers, different activation record layout, different object code (different instructions), and program points mapping to different program counter values.

The differences in available registers, use of registers, and layout of activation records are essentially only differences in data representation. If we have sufficient information about how registers and temporary variables in activation records are being used at each program point, we can convert the call stack with all the relevant activation records to and from a machine-independent format.

The differences in program counter values for the same program point are slightly more troublesome. To move program counter values, we must compute program counter values on the destination processor that correspond to the program counter values on the origin processor. However, there may be program counter values for one type of machine-dependent code that do not have corresponding program counter values in a different type of machine-dependent code. Even given the assumption that the same transformations and optimizations have been performed on the different types of code, non-correspondence may happen when certain operations are non-atomic on some processors. For example, unlinking an element from a doubly linked list is atomic on the VAX processor but requires multiple instructions on the SPARC processor.

One way to avoid this problem is to simply prevent the mobility layer of the runtime system from ever seeing such program counter values. We say that the critical program counter values are made invisible and that the remaining values are visible. Such restriction of visibility be achieved in that multiple ways.

The Trellis/Owl system [SCB+86] permitted transfer of control to the runtime system at any time (*e.g.*, by interrupts), but if the transfer of control happened in a critical region, the top layer of the runtime system would execute by interpretation the necessary number of instructions to exit the critical region before calling the lower layers of the runtime system that manipulated thread states.

Thus Trellis/Owl simply avoids having to deal with seeing program counters inside critical sections.

We may consider any program counter value to point into a critical region, if the program counter value does not have corresponding program counter values for (at least) one different architecture. By always interpreting instructions until reaching a "safe" program counter value, the thread state manipulating parts of the runtime system (*e.g.*, the parts implementing thread mobility) will never see a program counter value that does not have corresponding program counter values in all other types of machine-dependent code.

The Emerald system relies on its compiler to generate code that transfers control to the runtime system instead of having the runtime system preempt the threads [Jul89]. Control can only be transferred to the runtime system at suitable chosen points: at system calls, operation invocation entry (procedure/function calls), and at the bottom of loops. If the same transformations and optimizations have been performed on all types of machine-dependent code then choosing these points ensures that the runtime system only sees program counter values that have corresponding program counter values for different types of machine-dependent code. In Emerald, this technique is also used to provide the garbage collector with well-defined states for easy pointer identification ([JLHB88, JJ92, Juu93]).

We can enumerate all the program counter values that have corresponding program counter values in different types of machine-dependent code. The number of such program counter values will be the same on all processor types. We can perform this enumeration such that it is consistent across the different types of processors. The assigned numbers then uniquely specify program points independent of the type of code being executed. These numbers can therefore be used as machine-independent specifications of program points.

We use the metaphor *bus stops* to describe the enumerated program points. There may be many different sequences of basic operation that can be performed between bus stops, we do not really care about all these different ways, as we only ever stop at the bus stops. A compiler is free to reorder and optimize between bus stops.

Bus stops can be considered safe migration points where any native code generator must ensure that the thread state can be translated to and from a machine-independent form. Given a set of bus stops, the code generator is free to optimize code between bus stops in any way, as the optimization transformations are not visible to the runtime system In this respect, bus stops are related to the *synchronization points* of ANSI C [Ame89].

Compiler support is necessary to generate both the information needed to describe machine-dependent use of registers and temporaries in activation records and the bus stop information. *No change to the generated code is necessary.*

A considerable amount of data conversion has to be performed by the runtime system when moving machine-dependent thread state Mobility of machine-dependent thread state among heterogeneous processors is likely to be more expensive than mobility of machine-independent thread states. However, the advantage of converting between machine-dependent and machine-independent formats is that native code performance can be achieved while thread states are not being moved. The solution therefore appears acceptable when intra-node runtime performance is more important than thread mobility performance Even when thread mobility performance is important, our unoptimized implementation of heterogeneous thread mobility is acceptable in many cases: it takes only 60% longer to perform a thread move as on the version supporting homogeneous mobility.

In this section, we have presented our concept of bus stops and described how this technique can be used to achieve heterogeneous thread mobility while allowing for compiler optimizations between bus stops. In the next section, we discuss allowing compiler optimization across bus stops.

| abstract | code1 | code2 |
|---|---|---|
| `op1;` | `op1;` | `op2;` |
| `op2;` | `switch();` | `op5;` |
| `op3;` | `op2;` | `switch();` |
| `switch();` | `op3;` | `op4;` |
| `op4;` | `op4;` | `op1;` |
| `op5;` | `op5;` | `op3;` |
| `op6;` | `op6;` | `op6;` |

Figure 3. Bridging Code Example: Example of how a machine-independent code sequence (abstract) may be optimized in two different ways by code motion (code1 and code2)

### 2.2.2 Differences in Optimization

Orthogonal to the issue of machine-dependent vs. independent code, is whether or not the code at the source processor is transformed and optimized in exactly the same way as the code at the destination processor. Even given homogeneous machines, possible differences in transformations include code motion to change lifetimes of values, strength reduction, etc.

Thread mobility is fairly easy using bus stops, if no visible program counter value points into the code in question. So one way to allow mobility among differently optimized codes is to only permit code transformations between visible program points. However, this is likely too restrictive, allowing only small peep-hole optimizations. Modern compiler techniques often result in more general code reorganization. In this section we describe how to enable thread mobility in the presence of general code transformations between the source and destination codes.

In contrast to the techniques described in the previous section, the techniques described in this section are not backed up by a prototype implementation demonstrating the validity of the techniques. However, the issues are worth considering, and we believe our suggested techniques to be applicable.

Many different types of program transformations and optimizations exist. For now, we will only consider various types of code motion transformations, data changing optimizations such as strength reduction in loops, and RISCifying transformations, replacing a complex operation with several simpler ones[3]. Generalizations are possible but are outside the scope of this paper.

By code motion we mean reordering of instructions that may occur on a given path through the program. From the perspective of thread state mobility, code motion may have the effect that instructions are moved around a potentially visible program point. The instructions may have side effects, so it is important that the instructions are executed exactly once. One way to overcome code motion differences between different compiled instances of the code is to build *bridging code* between the origin and destination instances of the code. The different instances of code can be viewed as the super-blocks of Trace Scheduling [Fis81]; the bridging code is then equivalent to the entry paths to and the exit paths from the super-block, and the bridging code can be constructed using similar techniques.

**Example 2** Consider the code sequences shown in Figure 3. The leftmost code sequence is the unoptimized code sequence handed to the backend of the compiler. The two other code sequences are examples of how the original code sequence can be modified by code motion transformations.

Assume that code1 is part of the code for an object to be moved, and the program counter value corresponding to the `switch()` operation is visible. The program counter may be visible, if `switch()`

[3] RISCification is common in, *e.g.*, compilers for the Pentium processor [Int94] where only a subset of simple instructions may be executed simultaneously with other instructions in the processor's other execution pipeline

```
code1                bridge          code2
op1;                                 op2;
switch();                            op5;
op2;       ------>   op2;            switch();
op3;                 op4;            op4;
op4;                 op5;            op1;
op5;                      ------>    op3;
op6;                                 op6;
```

Figure 4: Example of bridging code necessary to change from using the code sequence "code1" to using the code sequence "code2" at the "`switch()`" in "code1".

is either a procedure call or a system call. The object is to be moved to a processor where code2 is to be used instead of code1. Because of the code motion transformations, there is no direct correspondent in code2 to the visible program point in code1 (the program point is not a bus stop). Therefore, we must generate bridging code to overcome the differences Figure 4 illustrates the bridging code necessary to overcome the differences between the code sequences code1 and code2.

Operation `op1` has already been executed at the time control reaches the `switch()` operation in code1. There is a bus stop at operation `op6` in both codes, at which point we can start executing the instructions from code2. Before doing so, we have to ensure that operations `op2`, `op3`, `op4`, and `op5` are executed exactly once. Operation `op3` can be executed in code2. To execute the remaining operations, we *generate* a new code fragment containing `op2`, `op4`, and `op5`. After `op5`, the code fragment jumps to `op3` in code2. The program counter value at `switch()` in code1 is translated to the program counter value indicating `op2` in the new code fragment. □

Code motion can be implemented by a very small set of primitive operations on control flow graphs. Assume that the optimization phase of the compiler is given an initial control flow graph. We can then duplicate the control flow graph and create links between identical program points in the two versions of the graph If the optimization phase of the compiler optimizes one of the two versions of the control flow graph by the primitive code motion operations, each such operation can automatically generate the bridging code in both directions between the original and the optimized control flow graph. If the primitive code motion operations are all reversible, reversing the sequence of code motion operations and performing the reverse of each code motion operation on the optimized version of the control flow graph will yield the original control flow graph.

Given the optimized code, the original code, the bridging code between the two, and a specification of how to construct the bridging code from the original code (in terms of primitive code editing operations), it is possible to implement thread state mobility among processors executing code that has been subject to different code motion transformations Assume that a thread has been temporarily halted at a certain program point in the optimized code on the originating processor. The program point can be specified by two components 1) how to create the bridging code to the original, unoptimized code, and 2) the point in the original code reached by the bridging code At the destination processor, the bridging code from the visible program point (at the source node) to the original code can be constructed using the set of primitive editing operations from 1) We then append to that, the bridging code from the reached program point in the original, unoptimized code to the optimized code on the destination processor. The result is bridging code from the optimized code on the origin processor to the optimized code on the destination processor. By making the thread start executing the bridging code on the destination processor, we ensure that each operation is executed exactly once (as it should be) and that the thread eventually will execute optimized code on the destination processor if it is not migrated while still executing the bridging code.

**Example 3** The bridging code shown in Figure 4 could be generated by first generating bridging code from code1 to "abstract" shown in Figure 3 and then generating bridging code from "abstract" to code2 The bridging code from code1 to "abstract" consists of operations `op2` and `op3`. Bridging from "abstract" to code2 removes `op3` and inserts `op4` and `op5` in the bridging code. □

The thread state may, of course, be moved once more before it has finished executing the bridging code. This is not a problem, if we either avoid bus stops in bridging code or, more generally, if we retain the description of how the bridging code was constructed (in terms of primitive editing operations). Bridging code from bridging code can be constructed the same way as the bridging code between the original and the optimized control flow graphs.

Strength reduction in loops is an optimization that not only requires transformation of code but also requires transformation of data in the thread state. If the compiler provides a complete description of the transformation, we can convert the thread state data as necessary while constructing bridging code between the different types of optimized code.

Instruction selection is a very fundamental operation during code generation. Assume the compiler backend is given a control flow graph representation of the program. Some of the operations in the control flow graph may perhaps be implementable by single machine code instructions on the processor we generate code for Other operations (*e.g.*, unlinking an element from a doubly linked list) may not be implementable by a single machine instruction. These more complex operations may be replaced with a sequence of other operations, which may be implemented by single machine instructions. Replacing a complex instruction with several simpler instructions may also be desirable for RISCification purposes.

In the context of thread state mobility, it is problematic if a visible program counter value indicates a program point where some, but not all, of the instructions resulting from the instruction selection for a single operation have been executed. If the operation that results in multiple instructions on the originating processor only results in a single instruction (or in multiple different instructions) on the destination processor, there is no direct correspondence between operations in the different codes. Again, a possible solution is to generate bridging code based on instruction selection information generated by the compiler.

As mentioned in the beginning of this section, the issue of mobility of threads between (processors with) code optimized in different ways is orthogonal to the issue of mobility between heterogeneous processors. They are independent system dimensions.

### 2.3 System Dimensions and Compiler Support

There are three system dimensions that are important when considering object and thread mobility among heterogeneous processors:

1. Machine-dependent vs. machine-independent data,
2. Machine-dependent vs machine-independent code,
3. Existence or non-existence of codes that visibly have been transformed or optimized differently.

If data is represented in a machine-dependent form at any time on any type of processor, compiler support is required to enable object mobility. The type of information required is typically limited to structure layout and the types of the values kept in the structure slots Such information is usually also required by symbolic debuggers. Typically, only a small amount of extra information is required to support mobility.

To enable thread mobility when threads are executing machine-dependent code, additional compiler support is required. The usual debugging information will typically be sufficient to describe most of the data component of the thread state. Extra compiler support may be necessary to describe the use of temporary values at each bus stop. Compiler support is also necessary to associate program counter values with bus stop numbers. The necessary additional compiler support is similar in extent to the usual debugging information.

To enable thread mobility when the executed code at origin and destination processors may be optimized in different ways, extensive compiler support is required. The backend of the compiler must generate information completely describing the transformations performed during code generation. Also, the backend of the compiler must be tied closely to the runtime system for the purpose of dynamically generating the necessary bridging code. Whereas the first two system dimensions only require compiler support to enable the runtime system to transform the machine-dependent format into a machine-independent format, the possible existence of codes optimized in different ways requires the runtime system to be able to invoke parts of the compiler at runtime.

## 3 Implementing Heterogeneous Mobility in the Emerald Prototype

We have an Emerald prototype implementation of our ideas which shows that object and thread mobility is possible among heterogeneous processors, even if the processors operate on machine-dependent data and have machine-dependent thread state. The executed code on all types of processors has, however, been subject to exactly the same optimizations, so the prototype only demonstrates the solution techniques for the first two of the three system dimensions identified in the previous section.

The prototype is an extension of the Emerald programming system, which originally supported object and thread mobility among homogeneous processors [JLHB88, Jul89]. In the following subsections, we will discuss the goals for our prototype implementation (Section 3.1), the features of the original Emerald implementation that are relevant for this paper (Section 3.2), what changes we had to make to the Emerald compiler (Section 3.3), changes to the compilation process (Section 3.4), and what changes we had to make to the Emerald runtime system (Section 3.5). In the final subsection (Section 3.6) we describe our experience with the prototype implementation.

### 3.1 Design Goals for Our Implementation

The original goal of the Emerald projects was to demonstrate that object and thread mobility was possible without sacrificing the runtime performance obtained by executing native machine code. This goal was achieved. The goal for our prototype is the same, with the addition that object and thread mobility must be possible among heterogeneous processors as well as among homogeneous processors.

For the prototype implementation, we did not want to focus on the performance of the runtime system when performing object and thread mobility. The purpose of the prototype was to prove possible the concept of native code thread mobility among heterogeneous processors. Also, we did not find it important to retain the existing performance of mobility among homogeneous processors. Previous work has shown that multiple protocols can be used for RPC in a heterogeneous environment to avoid the extra overhead of converting data to a machine-independent format (network format) when performing RPC between homogeneous processors [SC88]. The extra effort to do this was considered unimportant for demonstrating our points.

### 3.2 Features of the Original Emerald System

The original Emerald system supports both object and thread mobility among homogeneous processors not using distributed shared memory [JLHB88]. The processors are workstations connected in a local network. Fine-grained objects can be extracted from the address space of one processor and moved to another processor. All activation records describing invocations of methods in the moved objects are moved along with the objects, thereby implementing thread mobility.

The original Emerald system supports object and thread mobility on networks of homogeneous workstations of one of the following four types: VAXen running BSD-Unix or Ultrix, Sun-3s running SunOS, HP9000/300s running HP-UX, and Sun SPARC workstations running SunOS.

All data in Emerald consists of objects. Objects may refer to objects on other workstations. It is transparent to the programmer (modulo performance) whether or not a given object resides on the same processor as an object containing a reference to the object. In the implementation, references are object identifiers, OIDs, uniquely identifying objects regardless of their location. The only way threads can share data is by having references to the same objects. Since references are network transparent, threads may move independently of each other.

The Emerald compiler generates so-called templates which describe objects and activation records in sufficient detail to enable the runtime system to perform the necessary pointer swizzling and to update the distributed synchronization data structures when objects are moved from one processor to another. The templates do not distinguish between different forms of simple data, *i.e.*, integers, floating point values, strings, etc. The Emerald calling conventions include callee-saved registers, and the templates for activation records include sufficient information to distinguish registers holding pointer values from registers holding simple data and to find pointers in the callee-saved register area.

Only one template is used to describe activation records for invocations of a particular method. Both registers and slots in the activation record structure may be used to hold values of different types over the lifetime of the activation record, but the compiler ensures that a given slot will only hold either simple data or pointers throughout the lifetime of the activation record. The initial design of the Emerald system allowed for multiple templates for each activation record, each template being valid for a certain range of program counter values. Initial experiments found that multiple templates could be avoided by a combination of careful compiler design and the bus stops technique [Jul89].

Apart from the template information necessary for the runtime system to support mobility among homogeneous processors, the Emerald compiler also generates debugging information for use by a symbolic debugger. The debugging information identifies the exact locations and types of both global object variables and local variables.

Object code is encapsulated in code objects identified by OIDs. Code objects are immutable objects and can therefore by "moved" to another processor by duplication. Localization and mobility of code objects are performed by the same mechanisms performing localization and mobility of all other types of objects.

The Emerald runtime system only ever sees a restricted set of program counter values. From the runtime system's perspective, the object/user code is responsible for transferring control to the runtime system by system calls. The compiler is responsible for generating code that transfers control to the runtime system when necessary[4]. Transfer of control is performed by a system call.

[4] An interrupt handler can reset the stack limit pointer to indicate to the user code that control must be transferred to the runtime system. Checks for available stack space are performed by the user code at procedure calls and at the bottom of loops. The code sequence for method invocations must check for stack space availability anyway, so

The only program counter values visible to the runtime system are therefore at method invocations (the program counter values being return addresses stored in activation records), at the bottom of loops, and at system calls in the user code. Thus the original Emerald used a simple version of the bus stop technique

### 3.3 Changes to the Emerald Compiler

To enable thread mobility among heterogeneous processors, the compiler must generate information about bus stops, activation records layout, object layout, etc. It is not necessary to make any change to the generated machine code

The visible program counter values in the original Emerald system fulfill all the requirements of bus stops. A bidirectional mapping between program counter values and bus stop numbers is needed by the runtime system to convert program counter values to bus stop numbers and vice versa. To generate the bus stop mapping, we changed the backend procedures for generating the procedure call sequences and system call sequences to add entries to the mapping

While the debugging information generated by the Emerald compiler is sufficient to identify the location of all local variables, it does not specify which variables are dead or alive at a given program point Consequently, it does not specify which of potentially many variables are currently stored in a register or activation record slot shared by multiple variables Also, the template information does not indicate the number and types of temporary variables live at a given program point. The template information must therefore be augmented with information for each bus stop on which variables currently "own" shared locations and the number and types of temporary variables in use The code for adding an entry to the bus stop mapping captures and saves information about the number and types of live temporary variables and which local variables "own" shared registers or slots in the activation record at that program point

With one exception, these were the only changes necessary to the Emerald compiler[1]

The VAX processor can perform unlinking of a doubly linked list as an atomic operation. The Motorola 68000 processors (used in Sun-3 and HP9000/300 workstations) and SPARC processors cannot perform the unlinking as an atomic operation. As unlinking is used to implement monitors [Hoa74] in Emerald, a system call is required to ensure the atomicity of the unlinking operation The bus stops on all types of processors must be isomorphic to each other. We therefore have to add an entry to the bus stop mapping for each unlink instruction on the VAX processor, even though no system call is performed at that point in the code This bus stop is an *exit only* program point meaning that conversion from the bus stop number to the program counter value may be necessary, but not the other way around. Again, no changes are made to the code to be executed, we only need to generate template information on the side describing the program point

### 3.4 Changes to the Emerald Compilation Process

Aside from the changes to the compiler described in Section 3.3, we also made two crude changes to the compilation environment—short-cuts which in a production system would be replaced by more suitable compiler modifications.

The problem is that we need to have several compiled versions of a program one for each architecture For our prototype, we chose a primitive solution the programmer simply compiles the program once on each architecture. However, for the implementation to function correctly, it is necessary that the unique object identifiers, OIDs, are the same for all versions of the program

[1] most of the user code polls are "free" Roughly the same method is used to implement signals in Standard ML of New Jersey [Rep90]

In a homogeneous processor environment, the same object code can be used on all workstations, and it makes sense to assign object code OIDs. In a heterogeneous processor environment, only object code compiled for a specific type of workstation can be used. It is therefore no longer sufficient to describe object code with an OID and expect the mobility subsystem to fetch the correct code It is, however, desirable to retain the OIDs to describe the semantic content of a code object We want the OIDs to be the same for semantically equivalent code objects generated for different types of workstations. We therefore need to augment OIDs with another mechanism to distinguish code objects.

We did not add such a mechanism to our prototype In the prototype implementation, the programmer has to manually set the OID counter to ensure synchronization. While this is clearly impractical, the lack of this feature is not necessary to prove possible the concept of mobility among heterogeneous processors

In a fully functional implementation, the compiler has to maintain synchronization between OIDs for isomorphic object codes for different processors One possible method for doing so is to use a program database. When a file is compiled, the program is stored in the program database. When it is subsequently compiled on another processor, information from the program database is used to ensure that exactly the same OIDs are used Storing programs in a program database would also allow the runtime system to automatically invoke the compiler to generate processor specific object codes in case the programmer had not manually compiled the program for all the used types of workstation In our prototype, the programmer manually has to ensure the availability of object codes for all the types of workstations that an object may possibly move to, *i.e.*, compile the program once on each type of workstation

When a node receives an object for which it does not have any code, it searches for the code, first by checking on its disk, thereafter by searching the network We use NFS (SUN Network File System [SGK+85]) to create the illusion that the object code always resides in the local disk repository. When the kernel needs object code with a specific OID it thus gets the correct version from disk instead of from another (potentially heterogeneous) Emerald kernel.

### 3.5 Changes to the Emerald Runtime Kernel

The changes to the Emerald runtime kernel fall in two different categories: addition of procedures to convert to and from network format, and changes to the marshalling and unmarshalling code.

Conversion of ordinary data structures to and from network format is performed by a set of hand-written conversion routines The code is not optimized for speed but for easy of maintenance Composite data structures are converted by recursive descent of the structure. Depending on the processor type, 2–3 procedure calls are performed to convert a simple integer value to or from network format.

Conversion of program counter values to and from bus stop numbers are performed using the bus stop tables generated by the compiler New table lookup routines were necessary to perform the conversion, as we wanted to extract the associated information about local variables and temporaries at the same time.

Marshalling of data structures for transport over the network already existed in the original Emerald system. The marshalling code was instrumented to convert the data structures to and from network format as part of the marshalling process.

An additional layer of marshalling was necessary to convert activation records to and from a machine-independent format We invented a new activation record format and used that as the machine-independent format. The new activation record format stored all local variables in the activation record rather than in registers. The compiler generates sufficient template information to enable the runtime kernel to convert the machine-dependent activation records

to and from the machine-independent activation records. While conceptually simple, this transformation requires a fair amount of work and is easy to get wrong.

A machine-dependent activation record may require more or fewer bytes than the corresponding machine-dependent activation record, depending on the nature of the procedure (method) in question. This created an extra problem at the destination processor, as the template and debugging information required us to translate the youngest activation records first. Since we could not know beforehand the size of the machine-dependent activation record stack (thread fragment), we could not perform an initially correct placement of the activation records in the allocated thread stack space. We therefore had to do a relocation of all activation records within the allocated stack space after the conversion to the machine-dependent format

### 3.6 Experience with and Performance of the Prototype

The prototype has been implemented. It can move objects and threads among all four types of workstations. The prototype has been tested on a small number of test cases It is cumbersome to ensure code object OID synchronization manually (as described in Section 3 4) In a production system, this problem is readily solved by a program database.

Our additions to the Emerald system have not affected the code generated by the compiler. Intra-node computations will therefore execute exactly the same instructions (and in the same order) on the original Emerald system and our enhanced Emerald system. The intra-node runtime performance (comparable to that provided by C compilers [Hut87] and on the SPARC architectures at times even better [Mar92]) should therefore in theory be *exactly the same* on both systems Measurements on both systems verify this trivially. Thus we have achieved one of our major goals: to provide heterogeneous thread mobility at the same time as preserving node-local performance.

In contrast, there is a rather large difference in performance for inter-node computations between the original Emerald system and the enhanced Emerald system Table 1 shows the relative costs of moving a small thread (13 variables in the fragment being moved) among hosts in the original and the enhanced system. The cost is given for moving a thread from a machine of one architecture to a machine of a different architecture and back, for a total of two thread moves. The time costs for moving a thread $X \rightarrow Y \rightarrow X$ are the same as the cost for moving a thread $Y \rightarrow X \rightarrow Y$. The SPARC machines are SparcStation SLC workstations with 20MHz SPARC processors and 16MB RAM running SunOS 4.1.3. The Sun3 machine is a Sun3/100 workstation with 16MB RAM running SunOS 4.1.1 We only have one Sun3 machine left, so we cannot include timings for Sun3↔Sun3 thread mobility. We no longer have two identical HP9000/300 workstations, so we have instead used the two we had that were most similar in terms of performance $HP9000/300_1$ is an HP Apollo 9000 Series 400 Model 433s with 72MB RAM. It is based on a 33MHz MC68040 processor. $HP9000/300_2$ is an HP Apollo 9000 Series 300 Model 385 with 64MB RAM. It is based on a 25MHz MC68030 processor. Both workstations are running HP-UX 9.0 We unfortunately lost our last VAX workstation shortly after getting the enhanced Emerald system up and running[5]. The performance figure for mobility among VAX workstations on the original Emerald system was obtained on VAXstation 2000 workstations running Ultrix 2.1. The workstations are all connected by a 10Mbit/s Ethernet network.

We attribute the greater part of the difference in performance to our inefficient implementation of the routines to convert simple data structures between machine and network format. An average of 1–2 calls of conversion procedures are performed for each byte being transferred over the network even when ignoring the cost of converting activation records to and from a machine-independent format. We believe the performance penalty of the enhanced system would be much less if we used more efficient routines for conversion of simple data structures.

We have not performed any experiments to clarify how much of the performance penalty is caused by the way the conversion routines are implemented. Therefore, we can only guess that we could reduce the performance penalty by 50% by using more efficient routines.

In summary, we do pay for the ability to do cross architecture invocations and thread and object mobility, but we retain the full advantage of native code compilation, namely that applications run at full speed locally.

| Systems | Original | Enhanced | Δ |
|---|---|---|---|
| SPARC↔SPARC | 40 ms | 63 ms | 57% |
| SPARC↔Sun3 | N/A | 122 ms | – |
| SPARC↔$HP9000/300_1$ | N/A | 52 ms | – |
| SPARC↔$HP9000/300_2$ | N/A | 57 ms | – |
| SPARC↔VAX | N/A | N/A | – |
| Sun-3↔Sun-3 | 65 ms | N/A | – |
| Sun-3↔$HP9000/300_1$ | N/A | 109 ms | – |
| Sun-3↔$HP9000/300_2$ | N/A | 113 ms | – |
| Sun-3↔VAX | N/A | N/A | – |
| $HP9000/300_1$↔$HP9000/300_2$ | 28 ms | 44 ms | 57% |
| $HP9000/300_1$↔VAX | N/A | N/A | – |
| $HP9000/300_1$↔VAX | N/A | N/A | – |
| VAX↔VAX | 79 ms | N/A | – |
| VAX↔VAX | 48 ms | 81 ms | 68% |

Table 1: Thread Mobility Timings: Time costs of moving a small thread (13 local variables in the fragment being moved) among various types of machines on the original Emerald system and the enhanced Emerald system. Each cost is for moving the thread from one machine to another machine and back for a total of two thread moves. Results marked N/A are not available because our last VAX has died and we have only one Sun-3 left. The last line showing the cost for VAX↔VAX thread migration is set apart, as the numbers are for migration of a smaller thread from a different program between more modern VAXen than VAXstation 2000 workstations running an earlier version of our prototype

### 3.7 Source Code

The source code for our prototype implementation is available on the SOSP'95 CD and via either of the addresses shown in Figure 5.

Note, that there are two implementations of Emerald: one as described here which we now call UC-Emerald and a newer but non-distributed byte-coded version called BC-Emerald which has a compiler written in Emerald. For further information refer to the addresses listed in Figure 5.

## 4 Future Work

Our prototype implementation demonstrates that object and thread mobility among heterogeneous processors is possible. Our implementation is however inefficient and impractical to use An obvious next step is of course to perform an optimized implementation, so the true overhead of converting object and thread states to and from network formats can be measured.

[5] We have unfortunately preserved only preliminary performance figures from when the enhanced system was running on VAX workstations

| | |
|---|---|
| **e-mail** | emerald@diku.dk |
| **FTP**: | ftp.diku.dk:/pub/diku/dists/emerald/ |
| **WWW** | http://www.diku.dk/research-groups/distlab/emerald/ |

Figure 5: For further information on heterogeneous Emerald, use either of the above addresses.

This work also suggests two possible directions for further research.

1. Automatic generation of bridging code from transformation descriptions.
2. Use of thread state transformations in other programming tools

To our knowledge, there is no implementation demonstrating that it is possible to convert (move) a thread fragment based on code optimized in one way to a thread fragment based on code optimized in a different way. We have suggested one technique to do this based on the generation of bridging code between machine-dependent and machine-independent codes, but we have no prototype implementation to demonstrate the concept A prototype implementation demonstrating that thread mobility is possible in the presence of differently optimized code for the same methods would be the final proof that native code thread mobility among processors with heterogeneous code is possible with sufficient compiler support.

It may not be practical to generate bridging code at compile time between all visible program points in the machine-dependent and machine-independent object codes. Bridging code could instead be generated on demand. It should be possible to generate the bridging code from a description of the transformations performed by the backend.

The technique of generating bridging code when moving machine-dependent thread state may find uses in other programming tools The use of bus stops already has uses in debugging tools (see the related work section).

The automatic generation of bridging code may also be useful for debugging optimized code. A common problem with debugging optimized code is that there is no one program counter value that corresponds exactly to a given point in the program text In the situation where a programmer wants to insert a breakpoint at a given point in the program text for which there is no one program counter value, the debugger could insert bridging code in the optimized program to ensure that the desired thread state was created The breakpoint would then be in the bridging code Using bridging code to obtain the desired program points for inserting breakpoints does not solve the problem of debugging optimized code, as it cannot be used to recreate thread states in post-mortem debugging.

## 5 Related Work

There is work on progress on other implementations demonstrating that mobility of object and native code threads among heterogeneous processors is possible. There is also work related to the techniques discussed in this paper.

### 5.1 Mobility among Heterogeneous Processors

Jan Kølander at DIKU has modified the data conversion layer of our prototype implementation. He used the ISODE suite of tools to base the networking layer in Emerald on the ISO protocols and automatically generate code for conversion of data to a network format. He also worked on an extension of the ISODE tools to describe thread states in a machine-independent manner for the purposes of transferring a thread state over the network [Køl94].

Peter Smith at the University of British Columbia, Canada, is investigating techniques to implement a heterogeneous migration package to be used primarily with C but which should be usable for other languages [Smi95]. His system enables mobility of entire (native code) processes among Sun-3 and Sun-4 workstations. The programs must all be written in a type-safe subset of C and must be compiled by a special C compiler The codes for a single program compiled for heterogeneous processors have all been subject to exactly the same transformations His *preemption points* are similar to our *bus stops*.

Søren Brandt at the University of Aarhus, Denmark, is planning a migration and checkpointing package [Bra94] for the BETA programming language [MMPN93]. The package should enable checkpointing and migration of objects, including process objects, among heterogeneous workstations running BETA. This work is still in the design stage

### 5.2 Relevant Techniques

IN the Taos operating system [Pou91], all programs are compiled to a machine-independent byte code representation. At load-time, the byte code representation is compiled on the fly to native code. While we have no detailed information on the compilation process, we speculate it will be rather simple in order to be fast. Since the byte code version is available, it should be relatively easy to uncompile the code, identify bus stops, and translate the code dependent part of the activation records into a machine-independent format

If the byte code representation contains sufficient debugging information to translate the data areas and the data component of the machine state to and from a machine-independent form, it should be relatively easy to implement process mobility among heterogeneous processors in the Taos operating system

Java and TeleScript programs are also dynamically compiled from a machine-independent format to a machine-dependent format [Sun95, Tel95] It should be possible to implement mobility of Java and TeleScript processes among heterogeneous processors under the same conditions as for Taos processes. It is our understanding that the Taos, Java, and TeleScript compilers differ from the SELF compiler [HU94, Höl95] by not being very aggressive with respect to optimizations

The OSF's Architecture-Neutral Distribution Format (ANDF [TN94]) is a low-level machine-independent program representation. Programs are statically specialized to specific processors by *installers*. It should be possible to implement process mobility among heterogeneous processors using ANDF as the machine-independent format

The use of bus stops is already in use in several programming tools (*e.g.*, *interrupt points* in [DS84] and [Höl95] and *stopping points* in [Hen82]), and garbage collection (*e.g.*, the Emerald Garbage Collector [JLHB88, JJ92, Juu93], and *GC-points* in [DMH92]). In some systems, coarse-graind bus stops have been used for migration, *e.g.*, mobility in Eden was achieved by checkpointing an object, which dumped the object to stable storage, and subsequently moving the stored image and restarting it on another node [ABLN85, Bla85].

Mobility of threads/processes among heterogeneous processors relies on mapping information from a machine-dependent program format to a machine-independent program format and vice versa. Program slicing using optimized program representations also relies

on maintaining relationships between an optimized representation of the program and the original program representation (the source code or a parse tree) [Tip95, Ern94]. We expect future work in this area to develop techniques to maintain relationships between abstract values describing the unoptimized and optimized program representations respectively, even in the presence of data changing transformations such as strength reduction in loops Such techniques are likely to be directly useful for generating bridging code in the presence of data changing transformations.

Dynamic translation of optimized code to unoptimized code is performed in the latest version of the SELF system [Höl95]. They limit the number of optimizations they perform in order to be able to do the deoptimization.

A number of other systems offer native code process or thread mobility, *e.g.*, Sprite [Dou87] and DEMOS/MP [PM83], but they only support such mobility among homogeneous processors.

## 6 Conclusion

We have shown how object and thread mobility among heterogeneous computers can be implemented by converting both normal data and program state to and from a machine-independent format during move operations. We have presented the bus stop technique for handling migration of active threads. We have a prototype implementation in Emerald, which demonstrates the general technique for identically optimized code on all architectures. To the Emerald programmer, heterogeneity is transparent—the Emerald langauge is unchanged. Our prototype implementation retains the intra-node efficiency of the original Emerald, *i.e.*, node-local threads run at full native code speed We have also outlined techniques to enable mobility even for threads executing code generated with different optimization levels.

### Acknowledgments

The authors would like to thank Niels Christian Juul at DIKU for many interesting discussions about the Emerald system Thanks are also due to the many people that have encouraged us to write up this work (including the SOSP'93 audience at the work-in-progress section), as it would otherwise never had been documented Thanks also to Povl Koch at DIKU who contributed to the implementation during its early stages.

Thanks to the Danish Natural Science Foundation for providing equipment and programmer support at several points during the development of Emerald.

### References

[ABLN85] Guy T. Almes, Andrew P Black, Edward D. Lazowska, and Jerre D. Noe. The Eden System. A Technical Review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[Ame89] American National Standards Institute, Inc. Programming language — C, December 1989.

[BCL+87] Brian N. Bershad, Dennis T. Ching, Edward D. Lazowska, Jan Sanislo, and Michael Schwartz. A remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, August 1987.

[BHJ+87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 78–86, October 1986. ACM SIGPLAN Notices, 21(11).78–86, November 1986.

[Bla85] Andrew P. Black. Supporting distributed applications: Experience with Eden. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 181–193. Association for Computing Machinery, December 1985.

[BN84] Andrew D. Birrell and Bruce J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Bra94] Søren Brandt. Migration and checkpointing in BETA. Private communication, October 1994

[Car95] Luca Cardelli. A language with distributed scope. In *Proceedings 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 286–297, San Francisco, California, January 22-25, 1995.

[DMH92] Amer Diwan, Eliot Moss, and Richard Hudson Compiler support for garbage collection in a statically typed language. In *SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 273–282, San Francisco, California, January 17–19 1992.

[Dou87] Fred Douglis Process migration in the Sprite operating system Technical Report UCB/CSD 87/343, Computer Science Division, University of California, Berkeley, February 1987 A revised version of this paper appeared in the 7th International Conference on Distributed Computing Systems.

[DS84] Peter Deutsch and Alan Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings 11th Symposium on the Principles of Programming Languages*, Salt Lake City, Utah, 1984.

[Ern94] Michael Ernst. Practical fine-grained static slicing of optimized code Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA, July 1994.

[Fis81] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[Gib87] Phillip B. Gibbons. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering*, SE-13(1):77–87, January 1987.

[Hen82] John L. Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems*, 4(3), July 1982.

[Hoa74] C A.R. Hoare. Monitors: An operating system structuring construct. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hol95] Urs Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming* PhD thesis, Stanford University, March 1995

[HRB+87] Norman C Hutchinson, Rajendra K Raj, Andrew P Black, Henry M Levy, and Eric Jul. The Emerald programming language report. Technical Report 87-10-07, Department of Computer Science, University of Washington, Seattle, WA, October 1987

[HU94] Urs Holzle and David Ungar. A third-generation SELF implementation reconciling responsiveness with performance In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 229–243, Portland, OR, October 1994

[Hut87] Norman C Hutchinson. *Emerald: An Object-Based Language for Distributed Programming* PhD thesis, Department of Computer Science, University of Washington, Seattle, WA, January 1987.

[Int94] Intel Corporation. *Pentium Processor User's Manual*, 1994

[IOS94] International Organization for Standardization. *Information Technology — Abstract Syntax Notation One (ASN.1)*, February 1994.

[JJ92] Niels Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management, IWMM 92*, number 637 in Lecture Notes in Computer Science, pages 103–115, St. Malo, France, September 1992 INRIA, IRISA, and ACM Sigplan, Springer-Verlag

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1), February 1988 Appeared originally at SOSP'87.

[Jul89] Eric Jul. *Object Mobility in a Distributed Object-Oriented System*. PhD thesis, Department of Computer Science, University of Washington, Seattle, WA, January 1989.

[Juu93] Niels Christian Juul *Comprehensive, Concurrent, and Robust Garbage Collection in the Distributed, Object-Based System, Emerald* PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, February 1993. Available as Technical Report DIKU 93/1

[Køl94] Jan Kølander Implementation af OSI-protokoller i det distribuerede system Emerald vha. ISODE. Master's thesis, Department of Computer Science, University of Copenhagen, 1994. DIKU M.S. thesis no. 450 In Danish.

[Mar92] Jacob Marquard Porting Emerald to a SPARC. Master's thesis, DIKU, Department of Computer Science, University of Copenhagen, 1992.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language* ACM Press, 1993.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994. ISBN 0-201-63337-X

[PM83] Michael L Powell and Barton P. Miller. Process migration in DEMOS/MP In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 110–119, October 1983.

[Pou91] D. Pountain. Taos: An innovation in operating systems. *BYTE*, 16(1), March 1991.

[Rep90] John H. Reppy. Asynchronous signals in Standard ML. Technical Report TR 90-1144, Computer Science Department, Cornell University, August 1990.

[RTL+91] Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Andrew P. Black, Norman C Hutchinson, and Eric Jul Emerald. A general-purpose programming language. *Software — Practice and Experience*, 21(1):91–118, January 1991.

[SC88] Michael W. Strevell and Harvey G Cragon High-speed transformation of primitive data types in a heterogeneous distributed computer system In *The 8th International Conference on Distributed Computing Systems*, pages 41–45 IEEE Computer Society Press, June 1988.

[SCB+86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 17–29, October 1986 ACM SIGPLAN Notices 21(11):9–16, November 1986

[SGK+85] R. Sandberg, D. Goldberg, S Kleiman, D. Walsh, and B Lyon. Design and implementation of the SUN Network File System In *Proceedings of the Summer Usenix Conference*, 1985.

[Smi95] Peter Smith. Phd thesis proposal. The possibilities and limitations of heterogeneous process migration University of British Columbia, Canada , March 1995.

[Sun84] Sun Microsystems, Inc. *eXternal Data Representation Reference Manual*, September 1984

[Sun88] Sun Microsystems, Inc *Byteorder(3N)*, 1988 Unix manual page.

[Sun95] Sun Microsystems, Inc. The Java language: A white paper, 1995 Available from `http://java.sun.com/`.

[Tel95] TeleScript. Compilation of telescript programs Second-hand rumors, 1995.

[Tip95] Frank Tip. *Generation of Program Analysis Tools*. PhD thesis, Universiteit van Amsterdam, Institute for Logic, Language and Computation, 1995. Available as a book in the ILLC dissertation series (contact illc@fwi uva nl)

[TN94] Jens Ulrik Toft and Jens P. Nielsen. *Formal Specification of ANDF* DDC-International A/A, G. Lundtoftevej 1B, 2800 Lyngby, Denmark, January 1994 Document code 202104/RPT/19 issue 2.