To appear in the
Proceedings of PLDI'96, ACM SIGPLAN'96 Conf. on Programming Language Design and Implementation, May 1996, Philadelphia, PA

Efficient and Language-Independent Mobile Programs

Ali-Reza Adl-Tabatabai[1], Geoff Langdale[1], Steven Lucco[2] and Robert Wahbe[2]

[1]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 5213

[2]Colusa Software
1563 Solano Ave. MS-350
Berkeley, CA 94707

Abstract

This paper evaluates the design and implementation of Omniware: a safe, efficient, and language-independent system for executing mobile program modules. Previous approaches to implementing mobile code rely on either language semantics or abstract machine interpretation to enforce safety. In the former case, the mobile code system sacrifices universality to gain safety by dictating a particular source language or type system. In the latter case, the mobile code system sacrifices performance to gain safety through abstract machine interpretation.

Omniware uses software fault isolation, a technology developed to provide safe extension code for databases and operating systems, to achieve a unique combination of language-independence and excellent performance. Software fault isolation uses only the semantics of the underlying processor to determine whether a mobile code module can corrupt its execution environment. This separation of programming language implementation from program module safety enables our mobile code system to use a radically simplified virtual machine as its basis for portability. We measured the performance of Omniware using a suite of four SPEC92 programs on the Pentium, PowerPC, Mips, and Sparc processor architectures. Including the overhead for enforcing safety on all four processors, OmniVM executed the benchmark programs within 21% as fast as the optimized, unsafe code produced by the vendor-supplied compiler.

1 Introduction

The term *mobile code* describes any program that can be shipped unchanged to a heterogeneous collection of processors and executed *with identical semantics* on each processor. Currently, the most visible computer application requiring mobile code is executable content for electronic documents. While documents containing executable content have been around for at least two decades [46, 13], the combination of electronic documents with widely adopted network protocols [5] on the Internet requires *mobile* executable document content.

Because mobile programs are often untrusted, *safety* is an essential feature of any mobile code system. The system must maintain precise control over a mobile code module's access to the resources of its execution environment. We call the computer application that loads a mobile code module the *host*. To achieve safety, it is necessary for the mobile code mechanism to prevent a faulty or malicious module from corrupting host data or calling unauthorized host functions.

Our mobile code system, Omniware [28], uses software fault isolation (SFI) to enforce safety [42]. SFI enables mutually distrustful program modules to safely share an address space. Table 1 summarizes the Omniware performance for four SPEC92 programs; Section 4 provides detailed performance results. To support universal, efficient mobile code, we generalize the notion of a virtual machine to what we call a *software-defined computer architecture*. A software-defined computer architecture (SDCA) implements a set of virtual instructions, virtual memory management, and a virtual exception model. Our SDCA, the Omniware virtual machine (OmniVM), provides a segmented address space, enforces host-imposed permissions on access to this address space, and delivers an access violation exception to the module whenever it makes an unauthorized attempt to access a memory segment. Section 4 quantifies the overhead of using OmniVM to enforce write and execute protections on multi-page segments. Software fault isolation can also support efficient read protection and fine-grained access protection [42, 43].

program	Execution time relative to native			
	Mips	Sparc	PPC	x86
li	1.10	1.05	1.18	1.11
compress	1.04	1.02	1.23	1.02
alvinn	1.20	1.07	1.08	1.25
eqntott	1.20	1.04	1.35	1.06
average	1.14	1.05	1.21	1.11

Table 1: Execution time of translated code with SFI, relative to native code.

Omniware does not yet incorporate these capabilities[1].

SFI uses only the semantics of the underlying processor to determine whether a mobile code module can corrupt the host execution environment. SFI checks each unsafe memory access or indirect branch instruction at runtime to ensure that it does not violate access restrictions. The current Omniware system inlines these checks when a mobile program is loaded into the host address space and translated from OmniVM to native code (*load time*). This separation of programming language implementation from program module safety enables Omniware to use a radically simplified (RISC-like) virtual machine as its basis for portability [33].

This design choice has two advantages. First, it simplifies the implementation of compilers to OmniVM and translators from OmniVM. For example, we retargeted both `gcc` [17] and `lcc` [16] to OmniVM within two months. Second, the use of simple instructions gives the source language compiler more opportunity for optimization because more aspects (such as data layout) of the final code are defined by the compiler. Hence, a compiler can perform a great deal of machine-independent optimization (such as register allocation, constant folding, constant propagation, and strength reduction [2]) prior to module load time. This is important in many mobile code contexts such as Web pages where load time, and hence optimization during loading, must be minimized. Section 4 demonstrates quantitatively that compilers can substantially optimize OmniVM modules prior to load time.

The remainder of this paper is organized as follows. Section 2 describes in more detail the types of computer applications that can benefit from mobile code. Section 3 gives an overview of OmniVM and the tradeoffs in its design. Section 4 presents a detailed performance analysis of our system. We quantify the overheads introduced by translation and software fault isolation, and measure the effectiveness of optimizations performed by the translator. Section 5 compares our approach to mobile code with other mobile code systems and related projects.

[1]This document describes Colusa Omniware Beta version 1.68; later versions may have different features. This document does not provide an express or implied warranty for any Colusa product or license to any Colusa intellectual property.

2 Background

In the software industry, the need for mobile code is increasingly widespread. For example, distributed database systems [37] and file systems [30] require safe function shipping to achieve scalability. An e-mail client can ship a mail-filtering function to a server to reduce server bandwidth requirements. A file system server can ship a decompression function to a client to offload its processing.

Moreover, multi-platform operating systems, such as Microsoft Windows NT [31], when combined with network file systems, require either cumbersome management of processor-specific binaries or some form of mobile code. Similarly, distributed object-oriented database systems [4] use method invocation as a basis for data queries. In the absence of mobile code, these systems must manage heterogeneous binaries for each dynamically created data class. Because of these requirements, several languages for programming distributed systems, such as Orca [3] and Emerald [36], incorporate mobile code as a fundamental programming construct.

The mobile code systems introduced to date have used one of two methods for enforcing safe execution: abstract machine interpretation or language semantics. Abstract machine interpreters [12] trade performance for safety. A mobile code system based around an abstract machine consists of a compiler for some number of source languages coupled with an interpreter for the abstract machine. The interpreter manages the mapping between virtual resources used by the abstract machine and the physical resources of the host, preventing unauthorized access by checking that only valid mappings are used. This mechanism is effective and language-independent, but inherently slow.

A mobile code system can use language semantics to enforce safety by guaranteeing that a program can't affect resources that it can't name [38, 9, 40]. Though rare in current practice, it is possible to achieve good performance using this approach if the compiler intermediate representation retains type information [23, 39]. However, this approach works through restriction. For example, a strongly-typed intermediate language might promise through the type system that integer arithmetic will not be performed on a particular value, because the value has a pointer type. A virtual machine implementing this type system can check whether this promise is kept and reject programs that perform type-violating operations [18].

Hence, a mobile code system that uses language semantics to enforce safety sacrifices universality. This has two drawbacks. First, type-based mobile code systems can't implement type-unsafe languages such as C, C++, Pascal, Common Lisp, and Fortran. In the software industry, the vast majority of re-usable components, software libraries, and programmer expertise is in these languages.

Second, and most important, a mobile code infrastructure based on a particular type system imposes barriers to programming language innovation and experimentation. If a programmer invents a better type system, she can't simply deploy on the Internet a language embodying this type system, because a type-based infrastructure can only guarantee safety with respect to its existing type system.

3 The design of OmniVM

OmniVM is a RISC-based design that defines an instruction set, a register file organization, data formats, an exception model, and a segmented virtual memory model. The design of OmniVM reflects three goals:

1. **Portability.** OmniVM must be retargetable to a variety of target architectures. Moreover, OmniVM must be powerful and flexible enough to support the safe execution of any source language.

2. **Efficiency.** Translation of OmniVM must be fast, since in many applications of mobile code, translation speed is an important factor. Thus, a translator should not require expensive transformations. Such a translation scheme must still yield high performance object code on all target architectures.

3. **Ease of code generation.** OmniVM must be a simple target for a high-level language compiler.

3.1 RISC-based design

Portability across source languages is achieved with an architecture that resembles a typical RISC architecture. The OmniVM instruction set is rich enough to support any source language. OmniVM's load/store instruction set also makes it a simple code generator target.

The OmniVM architecture is designed to map directly onto a variety of target processors: each OmniVM instruction maps to one or more target machine instructions and each OmniVM register maps to a target machine register or memory location in the runtime environment. Since the majority of instructions map to a single target machine instruction, most of the generated object code has already been exposed to optimizations in the compiler. By delegating the bulk of optimizations to the compiler, the translator is left with the task of straightforward code generation. Hence, our system achieves efficiency in both the translation time and performance of generated target code.

The optimization opportunities that remain after translation are mainly machine-dependent optimizations. The measurements presented in Section 4 show that the bulk of speed-up opportunities are obtained by the machine-independent optmization performed by the compiler. These measurements shows that the design of our virtual machine instruction set effectively exposes target machine resources to compiler optimizations.

Number of Registers	Average Overhead
8	1.11
10	1.11
12	1.08
14	1.06
16	1.05
18	1.05
20	1.05
26	1.05

Table 2: Average execution time of mobile code relative to native Sparc code generated by `cc` for various OmniVM register file sizes.

Using a RISC-like virtual machine also yields competitive x86 code. A number of superscalar implementations of the x86 architecture provide a RISC core instruction set. For instance, Intel's Pentium [10] and Pentium Pro [20], AMD's K5 [35], and NexGen's Nx686 [21] processors are superscalar implementations of the x86 architecture, that can concurrently dispatch only RISC-like instructions [25]. An Intel application note describing instruction selection and scheduling for the Pentium processor, advises against selecting complex instructions and suggests using a load/store model of instruction selection [24].

3.2 Register file organization

The OmniVM has 16 integer registers and 16 floating-point registers. On the RISC targets, the OmniVM registers are mapped directly onto physical registers, while on the x86, some registers are mapped to memory locations. Since the newer implementations of the x86 architecture are optimized to efficiently execute instructions with memory operands, this strategy works well (see the performance measurements in Section 4). Table 2 shows that using fewer than 16 registers penalizes performance on the Sparc.

On RISC targets, the runtime system reserves some registers to efficiently implement SFI [42], to store environment information, and to preserve compatibility with the native application binary interface. On processors such as the PowerPC or Sparc, we could use as many as 23 OmniVM registers. However, Table 2 shows that Sparc performance does not improve significantly with the addition of more than 16 registers to the OmniVM.

3.3 Data formats

OmniVM defines the size of basic data types and supports integer types of byte, halfword, and word (8, 16, and 32 bits) and IEEE single- and double-precision floating-point types. We anticipate supporting 64 bit integer types and 64 bit addressing in future versions.

By defining the sizes of primitive data types, OmniVM enables the compiler to define the layout of aggregate data types and generate explicit address arithmetic code. This is important, because many optimizations such as common subexpression elimination, code motion, and strength reduction within loops, are effective on address arithmetic code.

OmniVM's basic data types are *endian-neutral*; the addressing of bytes within halfwords or words, and of halfwords within words is not defined. OmniVM provides data manipulation instructions to assure portability across machines with different byte orderings; these instructions are mapped to endian-dependent extract and insert instruction sequences of the target machine.

3.4 RISC instruction set enhancements

OmniVM generalizes sizes of address offsets and branch architecture. OmniVM provides memory access instructions with 32 bit immediate offsets. This choice guarantees that when an OmniVM translator encounters a memory access instruction, it has all the information necessary to generate good code for that instruction. On a CISC machine such as the x86, a single instruction will suffice. On a RISC machine, the translator typically generates one additional instruction for address calculation. Section 4 shows the overhead introduced through using large address offsets and how, by using a global pointer [11], we can eliminate the bulk of this overhead.

In contrast, if OmniVM were to restrict the size of address offsets, the compiler would generate additional address calculation instructions; further, an optimizing compiler might move these instructions across basic blocks. Hence, an x86 translator would be required to perform local or even global instruction combining to reconstruct the simplest instruction sequence for a given memory access.

OmniVM has general compare-and-branch instructions that branch on the result of a comparison between two registers or between a single register and an immediate value. These branch instructions are necessary for production of good code across a wide variety of branch models; for example, if compare results were placed in general purpose registers, it would be difficult to implement efficient conditional branches on architectures that use condition codes.

4 Evaluation

To evaluate the performance of mobile code based on OmniVM, we used `gcc` (version 2.4.5) to generate OmniVM instructions from a set of benchmark programs written in the C programming language; we used the OmniVM assembler and linker to generate OmniVM executable files for these programs. The executables generated by the linker are mobile code modules that are translated and executed by a host program that incorporates the runtime environment. The Omniware runtime environment includes both an OmniVM translator for the given target machine, and a set of library functions, such as memory management, threads, synchronization, and graphics that the host program can safely export to dynamically loaded Omniware modules.

We evaluated OmniVM translators and runtime environments on four different platforms: a Mips R4400 based SGI Indigo-2 machine running Irix 5.2, a PowerPC 601 based IBM RS/6000 running AIX version 3, a Sparc based SPARCStation 10 running Solaris 2.4, and a 90 MHz Intel Pentium based "Precision Pentium" machine running Microsoft Windows NT version 3.5.

Our translators include several optimizations. We have implemented local instruction scheduling in our Mips and PowerPC translators based on the algorithm described in [45]. We implement a global pointer in our Sparc translator and fill branch delay slots. On the x86, we perform only floating-point pipeline scheduling and peephole optimization. In addition, we have implemented instrumentation hooks in our translators so that we can gather information on the dynamic behavior of our benchmarks, such as instruction mixes. For the measurements in this section, no interprocedural optimizations were used.

We used four C programs from the SPEC92 suite for our evaluation: `li`, `compress`, `alvinn` and `eqntott`. The reference input files provided in the SPEC92 distribution were used as input data. We compiled these programs using both the vendor-supplied `cc` compiler and `gcc` (version 2.5.8 on Mips, Sparc and x86, version 2.6.3 on the PowerPC). For all compilers, we used the highest available level of intraprocedural global optimizations.

Our evaluation is organized as follows. In Section 4.1 we compare the performance of mobile code based on OmniVM with the performance of native code. In Section 4.2 we measure the performance improvements from performing optimizations in the translator. In Section 4.3 we measure the expansion in number of instructions due to differences between the OmniVM and host instruction sets, and due to software fault isolation.

4.1 Performance of mobile code

We compare the execution time of a dynamically loaded OmniVM executable with the execution time of object code generated by the native compiler. Tables 3 and 4 show execution times of translated OmniVM (both with and without SFI) relative to the execution times of native code generated by the host `cc` and `gcc` compilers. In the case of the three RISC architectures, the performance of safe mobile code based on OmniVM is virtually indistinguishable from the performance of native code generated by `gcc`. When compared to native

program	Execution time relative to native							
	Mips		Sparc		PPC		x86	
	SFI	no SFI	SFI	no SFI	SFI	no SFI	SFI	no SFI
li	1.10	0.91	1.05	1.02	1.18	1.08	1.11	1.10
compress	1.04	0.96	1.02	1.01	1.23	1.18	1.02	1.02
alvinn	1.20	1.09	1.07	1.03	1.08	0.97	1.25	1.22
eqntott	1.20	1.18	1.04	0.99	1.35	1.35	1.06	1.04
average	1.14	1.03	1.05	1.02	1.21	1.14	1.11	1.10

Table 3: Execution time of mobile code relative to native code generated by `cc`.

program	Execution time relative to native							
	Mips		Sparc		PPC		x86	
	SFI	no SFI	SFI	no SFI	SFI	no SFI	SFI	no SFI
li	1.11	0.92	1.05	1.01	1.04	0.94	1.09	1.09
compress	0.78	0.72	1.02	1.01	1.08	1.13	1.01	1.01
alvinn	1.12	1.01	1.08	1.02	1.36	1.21	1.09	1.06
eqntott	1.04	1.02	1.03	1.01	0.66	0.66	1.05	1.03
average	1.01	0.92	1.05	1.02	1.03	0.98	1.06	1.05

Table 4: Execution time of mobile code relative to native code generated by `gcc`.

program	Execution time relative to native							
	Mips		Sparc		PPC		x86	
	SFI	no SFI	SFI	no SFI	SFI	no SFI	SFI	no SFI
li	1.18	1.06	1.11	1.07	1.35	1.14	1.18	1.15
compress	1.04	0.84	1.18	1.16	1.28	1.23	1.09	1.07
alvinn	1.37	1.20	1.21	1.17	1.32	1.04	1.79	1.71
eqntott	1.08	1.06	1.24	1.21	1.35	1.35	1.22	1.16
average	1.17	1.04	1.21	1.15	1.33	1.19	1.32	1.27

Table 5: Execution time of mobile code without translator optimizations, relative to native code generated by `cc`.

program	Mips	Sparc	PPC	x86
li	0.98	1.01	1.14	1.13
compress	1.33	1.02	1.08	1.05
alvinn	1.07	1.01	0.80	1.38
eqntott	1.16	1.01	2.04	1.06
average	1.14	1.01	1.27	1.16

Table 6: Execution time of native code generated by `gcc` relative to native code generated by `cc`.

code generated by the `cc` compilers, mobile code is 10 - 20% slower.

There are four factors that contribute to performance differences between mobile and native `cc` generated code: (i) overheads due to software fault isolation, (ii) differences in the OmniVM and target instruction sets, (iii) better global optimizations in the `cc` compilers, and (iv) better machine-dependent optimizations in the `cc` compilers (e.g., instruction selection and scheduling). Because of the effects of cache misses and pipeline interlocks, it is difficult to quantify the contribution of each of these factors to execution times without simulating the pipeline and memory system of each architecture.

In Section 4.3 we quantify the effects of (i) and (ii) in terms of instruction counts and discuss techniques to alleviate these overheads. In the case of (iii), retargeting the `cc` compilers to OmniVM would result in faster mobile code, since mobile code would also benefit from reductions in path length due to better global optimizations.

We can measure the combined effects of (iii) and (iv) by comparing the native `cc` and `gcc` compilers. Table 6 shows the performance difference between native code compiled with `gcc` versus native code generated by `cc`. In general, the quality of code generated by the native `cc` compilers is better than `gcc`. The difference is greatest on the PowerPC (27%). We believe this is due mainly to better code selection and aggressive instruction scheduling performed by the `xlc` compiler. The PowerPC has a few features that are unusually challenging for code generators, specifically, auto-update addressing modes, branch-and-decrement instructions and multiple condition registers. Effective use of these features can result in substantial speed-ups [22], especially when the compiler performs global instruction scheduling [6].

We are currently enhancing our translators with global instruction scheduling and a framework for machine-dependent peephole optimizations. We expect these improvements to bring the performance of translated code on the PowerPC in line with that of the other two RISC processors. We also expect that these optimizations will improve performance of translated code on the x86, especially for pipelined implementations of the architecture, because Microsoft Visual C++ performs a number of complex peephole optimizations and instruction scheduling decisions for the Pentium that lead to its 16% performance difference from `gcc`.

Tables 3 and 4 also show the execution time overhead introduced by SFI. On all platforms, there is a performance penalty of approximately 10%. Other reports have investigated the effect of applying compiler optimizations to software fault isolation [42]. Based on these studies, we expect that Omniware's software fault isolation overhead will be cut to approximately 5% through these optimizations.

4.2 Benefits from translator optimizations

Table 5 shows that simple local instruction scheduling can substantially improve native code generated by OmniVM translators. This is encouraging, since mobile code applications will often require fast load times, making the use of global optimizations unattractive. This table shows execution times of mobile code relative to the execution times of native code generated by `cc`. Comparing Table 5 with Table 3 we see that the Mips, PowerPC, and x86 benefit greatly from instruction scheduling; we assume that this is because all three architectures offer machine-level parallelism that the instruction scheduler can exploit (the R4400 is superpipelined while the PowerPC 601 is superscalar). For example, the measurements for ALVINN on the x86 show the benefits of floating-point pipeline scheduling in our translator for the Pentium processor.

More importantly, instruction scheduling improves the performance of code translated with SFI, more than it improves the performance of code that has been translated without SFI. This is because instruction scheduling is able to hide some of the software fault isolation overhead within pipeline interlock cycles. Hence, the overhead of performing software fault isolation is alleviated by instruction scheduling. It is also interesting to observe from comparing Tables 5 and 3, that translator optimizations make up for some of the overhead introduced by SFI.

Although we do not perform instruction scheduling for the Sparc, performance on the Sparc is the most competitive with the native `cc` compiler. Sparc performance benefits from using a global data pointer (even though it has only a 13 bit offset for immediates) as well as using annulled branches. Since symbols are resolved during translation, our system does not pay the usual dynamic linking cost of setting and restoring a global pointer on each function call. We expect better performance from our Mips and PowerPC translators once we implement a global pointer in these translators; the numbers presented in Section 4.3 support this conclusion.

4.3 Instruction expansion

The charts in Figure 1 give a detailed view of the expansion introduced during translation from OmniVM instructions to native instructions, for the Mips and PowerPC architectures.

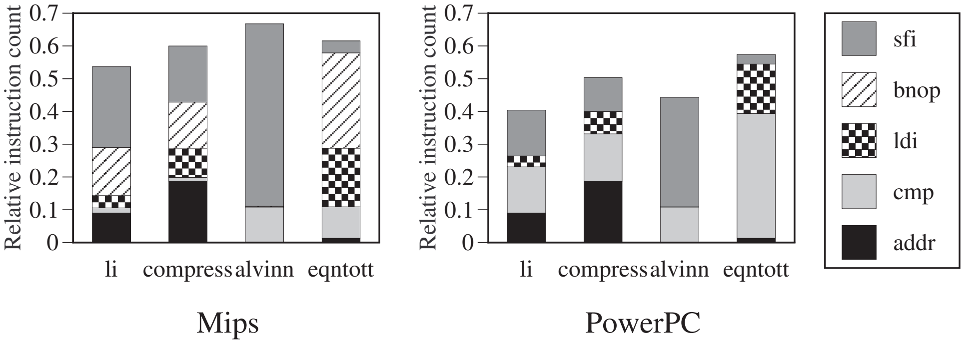

Figure 1: Expansion introduced by translation.

If all OmniVM instructions translated to a single native instruction, then there would be no expansion during translation. However, there are several situations where additional instructions are introduced during translation, and the charts in Figure 1 show the dynamic counts of these additional instructions relative to the number of OmniVM instructions executed:

- OmniVM includes addressing modes that translate to more than one instruction on these architectures. Expansion due to these additional instructions is labeled "addr" in the charts.
- OmniVM includes powerful conditional branches that sometimes translate to a compare and a branch on these architectures. Expansion due to these additional compares is labeled "cmp" in the charts.
- OmniVM has 32 bit immediates and thus additional instructions may be necessary to load an address or an immediate into a register when this immediate does not fit into the target architecture's instruction format. Expansion due to these additional instructions is labeled "ldi" in the charts.
- On the Mips, branches have a delay slot that must be filled. Expansion due to branch delay slots containing nops are labeled "bnop" in the charts.
- Additional instructions are inserted to enforce software fault isolation for unsafe store instructions. Expansion due to these additional instructions is labeled "sfi" in the charts.

These charts show how differences in the target architectures result in different overheads:

- The PowerPC executes substantially more compare instructions than the Mips. The reason for this is that most conditional branches in these programs involve a comparison against zero, which map to a single instruction on the Mips. The PowerPC must perform an explicit comparison for all conditional branches. Below, we discuss a few optimizations that can reduce this overhead on the PowerPC.
- The Mips has slightly more "ldi" overhead in `eqntott` and `compress`. This is because these programs have conditional branches involving comparisons against a constant. On the PowerPC, these constants fit into the immediate form of the compare instruction. The Mips has only a single compare with immediate instruction; if the branch condition does not match this compare instruction, the immediate must be first loaded into a register.
- The PowerPC executes fewer SFI instructions. The reason for this is that the PowerPC has an indexed addressing mode (i.e., register plus register) that allows the PowerPC SFI check sequence to be shorter than the check sequence for the Mips.
- Both architectures execute an equally significant number of addressing mode overhead instructions in `li` and `compress`. The OmniVM indexed addressing mode maps one-to-one on the PowerPC but requires an additional add instruction on the Mips. Since there is no difference in the addressing mode expansion between these two architectures, indexed mode addressing does not seem to be important for these programs. Both architectures execute many addressing mode overhead instructions that can be eliminated with a global pointer.
- Even though the scheduler attempts to fill branch delay slots on the Mips, there are still many branch nops that remain.

4.4 Discussion

The performance measurements presented in this section demonstrate that Omniware can achieve excellent mobile code performance. Omniware's overhead of only 10-20%

makes it an order of magnitude faster than any other universal mobile code system, because other universal systems must rely on abstract machine interpretation to enforce safety [12, 32]. For many applications of mobile code, such as executable content for Internet documents, our current performance is sufficient. However, we plan to add the capability for aggressive optimization to our translators, including global optimizations, link-level (interprocedural) optimizations, and chip-specific transformations such as global instruction scheduling, instruction combination and the organization of code and data to fit cache capacity and layout [27]. By adding these capabilities, we hope to make the Omniware system suitable for tasks such as general software distribution. In performing these optimizations, OmniVM translators have two advantages over typical compilers: not only does a translator have complete inter-procedural information (i.e., across module boundaries), it has exact knowledge of the target machine. Typically, compilers target an entire architecture family, not a particular processor implementation. Several studies suggest that we can significantly improve performance using this information [27, 44, 8].

In addition, our results suggest several simple steps towards this goal. First, implementing a global pointer can significantly improve performance. The performance improvement resulting from implementing a global pointer on the Sparc confirms this assertion. Second, the PowerPC is paying a significant overhead due to extra compare instructions, especially since these compare instructions have multi-cycle latencies and must be scheduled. Some comparisons against zero can be eliminated on the PowerPC by folding the setting of the condition codes into a prior arithmetic instruction. Moreover, the PowerPC branch and count instruction can fold an induction variable decrement, test against zero and branch instruction into a single instruction. Finally, SFI forms the foundation of our approach, but incurs an execution time overhead of approximately 10%. The overhead of SFI optimizations can be reduced using standard compiler techniques such as loop invariant code motion, as described in [42]. We have not implemented SFI optimizations and expect optimization will cut this overhead in half.

5 Related projects

Several projects have employed virtual machine architectures with low-level instruction sets that resemble the OmniVM instruction set. Designed for portable optimization rather than mobile code, Mahler [45] defines a virtual machine that abstracts over the details of several different Titan processor implementations. OmniVM differs from Mahler in the far wider range of architectures it supports, and the requirement for safe execution. Similarly, the Taos operating system [34] defines as its compiler target the Taos Virtual Processor, which, like Mahler, is an attempt to support multi-platform optimization.

Binary translation systems [1] address the problem of migrating existing native code from one platform to another. A similar approach is the idea of "fat binaries," where the compiler generates an object file containing multiple text sections – one for each of the target architectures. Neither of these methods address the issue of safety.

The ANDF [29] project is a recent attempt to standardize a universal intermediate language for software distribution [14]. ANDF's intermediate representation comprises typed expression trees. This representation is at a higher level than the OmniVM, and more work is required to translate it to native code. Thus, this representation is less suitable for applications where speed of translation is important, and will not benefit as much from compiler optimizations as OmniVM does. The OMI project [15] uses a similar approach.

Telescript [26] and Java [19] are two mobile code systems that achieve portability and safety by compiling to a machine-independent intermediate representation. Telescript enforces safety in its interpreter. Java depends on a type system for mobile code safety. Java's intermediate representation is tailored for fast interpretation by a stack machine [18], and, because it defers decisions such as data layout, requires more work than OmniVM to translate into efficient machine code. No performance evaluations have been released for Java, so it is difficult to evaluate the performance of the Java compiler or interpreter.

Some mobile code systems rely entirely on interpretation of source code at the host. Many scripting languages are in this category, including safe variants of Perl, Tcl [7] and Python [41]. These language-specific mobile code systems are useful for certain unstructured tasks such as parsing user input, but they require software distribution in source form and their performance is limited. A universal mobile code substrate such as Omniware provides a host program such as an Internet browser the capability for running any of these systems, without requiring that the host program statically incorporate a wide variety of interpreters. To provide a new interpreted language on the Internet, a programmer can write an interpreter in C or C++ and make it available as an Omniware module.

6 Conclusion

This paper described a mobile code system and its implementations on the Pentium, PowerPC, Mips, and Sparc processors. Including the overhead for enforcing safety, our current system can execute real C programs at execution speeds within 21% of the unsafe optimized code produced by the vendor-supplied compiler. Our evaluations suggest optimizations that can further improve performance. To our knowledge, Omniware is the fastest system for mobile code to date, and the first to efficiently implement safe, mobile

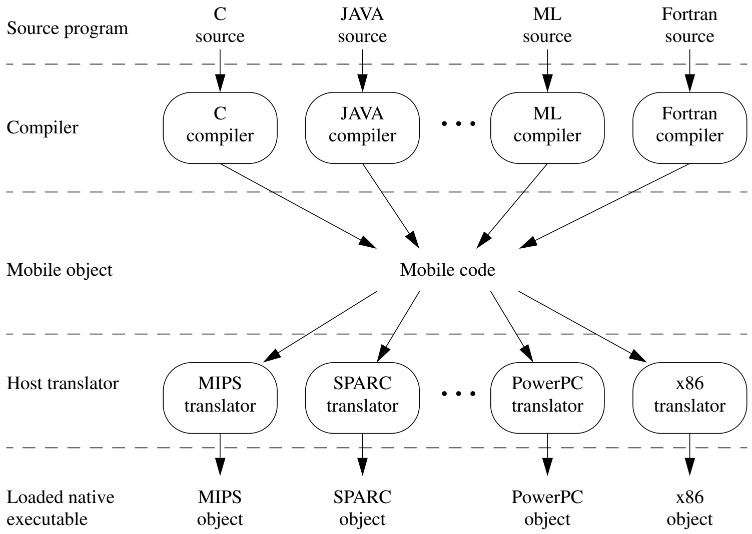

Figure 2: A universal mobile code substrate.

code in a language-independent way. Hence, as illustrated in Figure 2, we consider Omniware to be the first practical, universal substrate for mobile code.

References

[1] K. Andrews and D. Sand. Migrating a CISC computer family onto RISC via object code translation. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 213–222, October 1992.

[2] D. Bacon, S. Graham, and O. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.

[3] H. Bal, A. Tanenbaum, and M. Kaashoek. ORCA: a language for distributed programming. *SIGPLAN Notices*, 25(5):17–24, May 1990.

[4] J. Bennerjee, W. Kim, H. Kim, and H. Korth. Semantics and implementation of scheme evolution in object-oriented databases. In *Proc. ACM SIGMOD Conference*, pages 311–322, December 1987.

[5] T. Berners-Lee, R. Fielding, and H. Nielsen. HTTP/1.0 Internet Draft 04, October 1995. Internet Draft (work in progress).

[6] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 241–255, June 1991.

[7] N. Borenstein. EMail with a mind of its own: The Safe-Tcl language for enabled mail. In *IFIP Working Group 6.5 Conference*, May 1994.

[8] M. Burke and L. Torczon. Interprocedural optimization: eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, July 1993.

[9] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 language definition. *SIGPLAN Notices*, 27(8), August 1992.

[10] B. Case. Intel reveals Pentium implementation details. *Microprocessor Report*, pages 9–17, March 1993.

[11] F. Chow, S. Correll, M. Himelstein, E. Killian, and L. Weber. How many addressing modes are enough? In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–121, October 1987.

[12] K. Chung and H. Yuen. A tiny Pascal compiler. *Byte*, 39(9):58–64, September 1978.

[13] Compton's Interactive Encyclopedia, 1995.

[14] J. Strong et. al. The problem of programming communication with changing machines. *Communications of the ACM*, 1(8):12–18, August 1958.

[15] M. Franz. *Code-Generation On-the-Fly: A Key to Portable Software*. PhD thesis, Swiss Federal Institute of Technology Zurich, 1994. Diss. ETH No. 10497.

[16] C. Fraser and D. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, Oct 1991.

[17] Gcc, 1994. Free Software Foundation.

[18] J. Gosling. Java intermediate bytecodes. In *ACM SIGPLAN Workshop on Intermediate Representations (IR '95)*, pages 111–118, January 1995.

[19] J. Gosling and H. McGilton. The Java language environment: A white paper, 1995. Sun Microsystems, Inc.

[20] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, pages 9–15, February 1995.

[21] L. Gwennap. Nx686 goes toe-to-toe with Pentium Pro. *Microprocessor Report*, pages 1–10, October 1995.

[22] C. B. Hall and K. O'Brien. Performance characteristics of architectural features of the IBM RISC System/6000. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 303–309, April 1991.

[23] R. Harper and P. Lee. Advanced languages for systems software: The Fox project in 1994. Technical report, School of Computer Science, Carnegie Mellon University, January 1994. techreport CMU-CS-FOX-94-01.

[24] Intel. *Optimizations for Intel's 32-Bit Processors*. Application Note AP-500, Intel Corp., February 1994.

[25] K. Johnson. RISC-like design fares well for x86 CPUs. *Microprocessor Report*, pages 26–27, November 1995.

[26] S. Knaster. Magic Cap concepts, May 1995. General Magic, Inc.

[27] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[28] S. Lucco, O. Sharp, and R. Wahbe. Omniware: A universal substrate for web programming. In *Fourth International World Wide Web Conference*, December 1995.

[29] S. Macrakis. From UNCOL to ANDF: Progress in standard intermedia languages, 1993. Open Software Foundation.

[30] B. Noble, M. Price, and M. Satyanarayanan. A programming interface for application-aware adaptation in mobile computing. In *Proceedings of the Second USENIX Symposium on Mobile and Location-Independent Computing*, April 1995.

[31] Windows NT Workstation 3.51 Product Overview, 1995. Microsoft Corporation.

[32] J. Ousterhout. TCL: An embeddable command language. In *Proceedings of the 1990 Usenix Winter Conference*, pages 22–26, January 1990.

[33] D. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, January 1985.

[34] D. Pountain. Parallel course. *Byte*, 19(7):53–60, July 1994.

[35] M. Slater. AMD's K5 designed to outrun Pentium. *Microprocessor Report*, pages 1–11, October 1994.

[36] B. Steensgaard and E. Jul. Object and native code thread mobility among heterogeneous computers. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[37] M. Stonebraker and G. Kemnitz. The POSTGRES next-generation database management system. *Communications of the ACM*, 34(10):78–92, October 1991.

[38] R. Sweet. The Mesa programming environment. In *Proceedings SIGPLAN Symposium on Language Issues in Programming Environments*, pages 216–229, July 1985.

[39] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Languages Design and Implementation*. ACM, May 1996. To appear.

[40] J. Ullman. *Elements of ML programming*. Prentice Hall, 1994.

[41] G. van Rossum. Python tutorial, October 1995. Online at: http://www.python.org/doc/tut/tut.html.

[42] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficent software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, June 1993.

[43] R. Wahbe, S. Lucco, and S. Graham. Practical data breakpoints: Design and implementation. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 1–12, June 1993.

[44] D. Wall. Global register allocation at link time. In *Proceedings of the 7th SIGPLAN Symposium on Compiler Construction*, pages 264–275, June 1986.

[45] D. Wall. Experience with a software-defined machine architecture. *ACM Transactions on Programming Languages and Systems*, 14(3), July 1992.

[46] G. Williams. Hypercard (personal toolkit). *Byte*, 12(14):109–117, December 1987.