Efficient and Language-Independent Mobile Programs

Ali-Reza Adl-Tabatabai Geoff Langdale Steven Lucco
Robert Wahbe

October 1995
CMU-CS-95-204

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes 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.
Our approach 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 used CISC instruction traces to augment the design of our RISC virtual machine, OmniVM. We measured the performance of the resulting system using a suite of four SPECMARK 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.

This work was supported by a software license grant from Colusa Software, Inc. Omniware is a registered trademark of Colusa Software.

Keywords: Mobile code, executable content, code generation, translation, intermediate representation, software distribution, software fault isolation

1 Introduction

The term *mobile code* describes any program representation 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 [40, 12], the combination of electronic documents with widely adopted network protocols [5] on the Internet requires *mobile* executable document content.

Because the program bits that come off a network wire 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 the *host* program and we assert that, to achieve safety, a mobile code system must guarantee the *black box behavior* of the module with respect to the execution environment provided by the host. Specifically, while it is in general undecidable whether a mobile code module will perform the actions it advertizes, it is completely 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, uses software fault isolation to enforce safety [36]. Table 1 summarizes the Omniware performance for four SPEC 92 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 not only a set of virtual instructions, but also virtual memory management and a virtual exception model. Our SDCA, called the Omniware virtual machine (OmniVM), provides a segmented virtual 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 [36, 37]. Omniware does not yet incorporate these capabilities.

SFI 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 Omniware to use a radically simplified virtual machine as its basis for portability. The use of SFI to enforce safety enables a key feature of the OmniVM design: the use of an enhanced RISC instruction set [28].

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` [16] and `lcc` [15] 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; it then compares our approach to mobile code with other mobile code systems and related projects. Section 3 gives an overview of the OmniVM design and of our OmniVM translators for the Mips, Sparc, PowerPC and x86 architectures. Section 4 presents a detailed performance analysis of our system. We quantify the overheads introduced by translation and software fault isolation, and

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

measure the effectiveness of optimizations performed by the translator.

2 Background

In the software industry, the number and type of systems that require mobile code is growing at an enormous rate. Behind this growth is a broad trend toward mature software development: an increase in the decomposition of software into re-usable components and hence in the design of extensible software systems. Software decomposition, when combined with a second trend, the widespread adoption of distributed computing, dictates the need for mobile code.

For example, distributed database systems [31] and file systems [25] 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. Further, multi-platform operating systems, such as Microsoft Windows NT [26], 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 [30], 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 [11] 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 environment, 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 physical resources that it can't name [32, 9, 34]. Though rare in current practice, it is possible to achieve good performance using this approach if the compiler intermediate form for the language retains type information [21, 33]. However, this approach works through restriction. In the context of mobile code, a type is essentially a promise not to do something. For example, a strongly-typed language intermediate form might promise through the type system that integer arithmetic will not be performed on 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 on one or more values [17].

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. Whether this infrastructure migrates to typesafe languages will be dictated by economics, education, and programmer preferences. In any case, such a migration would take decades.

Second, and most important, a mobile code system based on particular language semantics 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 the language-semantics based mobile code substrate can only guarantee safety with respect to its existing type system.

2.1 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 [38] defines a virtual machine that abstracts 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 [29] 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 but ignore safety. In the "fat binary" approach to software distribution, the compiler generates an object file containing multiple text sections - one for each of the target architectures.

The ANDF [24] project is a recent attempt to standardize a universal intermediate language for software distribution [13]. ANDF's intermediate form is comprised of 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 front-end compiler optimizations as OmniVM does. The OMI project [14] uses a similar approach.

Other mobile code systems that achieve portability and safety by compiling to a machine independent intermediate representation include Telescript [22] and Java [18]. Telescript enforces safety in its interpreter. Java depends on a type system for its safety. Java's intermediate representation is tailored for fast interpretation by a stack machine [17], and, because it defers decisions such as data layout, requires considerably 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 [35]. These language-specific mobile code systems are very useful for certain unstructured tasks such as parsing user input. 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 simply write an interpreter in C or C++ and make it available as an Omniware module.

3 Implementation

3.1 OmniVM

The Omniware virtual machine is an enhanced RISC architecture with 16 integer registers and 16 floating point registers. Table 2 shows that register pressure (currently measured only on Sparc) is not a significant source of overhead despite this pared-down register set. The Omniware runtime environment contains a translator that generates native code from OmniVM instructions.

Number of Registers	Average Overhead
8	11.3%
10	10.5%
12	8.0%
14	6.0%
16	5.0%
18	5.0%
20	5.0%
26	4.6%

Table 2: Effect of register pressure on Sparc overhead

The OmniVM design reflects three goals: efficiency, portability and ease of code generation. We avoided complex instructions in favor of sequences of simple instructions, unless a complex instruction was necessary for portability or safety, or was frequently executed and impossible to decompose into a sequence of simple instructions.

Our main departures from RISC design principles are complex addressing modes and compare-and-branch instructions. Complex addressing modes are necessary to get good performance on CISC architectures without requiring global optimization in the OmniVM translator. We instrumented several large 32-bit programs (including `gcc` version 2.6.3) compiled by Microsoft Visual C++ on Windows NT to collect instruction mix statistics. These statistics show that Visual C++ makes frequent use of complex addressing modes in implementing C++ programs on x86 processors. For these measurements, we classified as RISC instructions simple branch instructions, arithmetic instructions involving registers, push and pop instructions and arithmetic instructions involving memory operands with simple addressing modes. The latter were included in the RISC category because they can be readily synthesized.

We found that 79% of all instructions executed could be replaced by RISC instructions or synthesized into RISC instructions using only basic block instruction combination. Of the remaining 21% of dynamically executed instructions, only two categories contributed over 3% to the instruction mix: block move or compare instructions (4%) and complex addressing modes (13%). We chose not to add a block move instruction to OmniVM because the preponderance of block move and compare instructions were found in the C library string routines (`memcpy, strcpy, strcmp`), which our translator can supply specially.

Complex addressing modes, however, can not be readily synthesized or encapsulated in a library routine; they require global optimization. For example, the OmniVM instruction set includes a symbol+register addressing mode; the symbol will be resolved to an address by the Omniware linker. RISC processors like the Sparc and Mips require two instructions to implement this addressing mode. If we take this addressing mode away from a compiler, but leave the compiler the indexed addressing mode (register+register), a compiler will place the value of the symbol in a register and then use the indexed addressing mode. However, processors such as the Intel x86 implement complex addressing modes in a single instruction. Since a translator to the x86 would require global optimization to re-combine the two instructions, we chose to include the common complex addressing modes in the OmniVM instruction set.

Section 4 shows that this has a small negative consequence for RISC translation targets, because, if a symbol is used in more than one memory access, global optimization may be required to eliminate the common sub-expression that loads the symbol into a register. However, Section 4 also shows that by using a global pointer [10] on the Sparc, we can eliminate the bulk of this overhead. To optimize for use of a global pointer, the translator generates, for every access to static data, code that addresses the data by offset from a register

pointing into the static data area. We could have chosen to have the compiler generate instructions that use a global pointer, but then we would have prejudiced the OmniVM code against CISC machines that have complex addressing modes but few registers to spare for a global pointer (such as the Motorola 680x0 series and the Intel x86). Therefore, our OmniVM design includes the CISC-like feature of complex addressing modes (specifically symbol+offset, symbol+register, and symbol+register+offset) because it gives both RISC and CISC translators the opportunity to generate good code.

Similarly, powerful branch instructions are necessary for production of good code across a wide variety of branch models; for example, if our compare results were placed in general purpose registers, it would be difficult to implement conditional branches on architectures that use condition codes. To ease global analysis, control flow and relocation information is readily available from the object file[19]. The conservative size of the OmniVM register set leaves us with enough registers to easily implement code transformations such as SFI.

OmniVM defines data types, sizes and layout, and provides instructions to assure portability across machines with different underlying representations of these types. We support integer types of byte, halfword and word (8,16 and 32 bits) and IEEE single- and double-precision floating point types.

3.2 Optimization

The basis for OmniVM's excellent performance is that most conventional compiler optimizations can be done on the machine-independent form. The combination of machine-dependent and machine-independent optimizations typically speeds up programs by more than a factor of 2. For example, the average speedup for the SPEC 92 benchmarks on a Mips Decstation 5000/240 is 2.2 times. With our four benchmark programs, we obtain the bulk of this speed-up from front-end machine-independent optimization. The next section shows that the design of our virtual machine instruction set effectively (although abstractly) exposes target machine resources to the front-end.

We also enhance performance by performing basic-block optimization during OmniVM translation. Section 4 details the specific optimizations we applied to our benchmark programs. Most of the gains reported in Section 4 came from our local instruction scheduler [38].

We are currently adding global optimizations to our translators, although for applications such as Internet executable content, these optimizations will probably cost more in load time than they are worth in mobile module performance. In performing global optimization, OmniVM translators have two advantages over typical compilers; not only do we have complete inter-procedural information (enabling link-time optimizations), we have exact knowledge of the target machine. Typically, compilers target an entire architecture family, not a particular machine. Several studies suggest that we can significantly enhance performance using this information [23, 39, 8].

4 Results

To evaluate the performance of mobile code based on OmniVM, we have retargeted `gcc` (version 2.4.5) to generate OmniVM assembly code, and have developed an assembler and linker that generate OmniVM object files and executables. The executables generated by the linker are mobile code modules that are translated and executed by a host program that incorporates the Omniware 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 have implemented OmniVM translators and runtime environments for 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

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`

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 [38]. In our Mips and Sparc translators, we fill branch delay slots using instructions preceding the branch. We implement a global pointer for all code (rather than just shared libraries) in our SPARC translator and fill delay slots with instructions from either the branch target (using an annulling branch when necessary) or from instructions preceding the branch. General local scheduling is not performed in the Sparc translator. 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, e.g., dynamic instruction mixes.

We use four C programs from the SPEC92 suite for our evaluation: `li`, `compress`, `alvinn` and `eqntott`. The reference input files provided in the SPEC92 distribution are used as input data. We generated direct native versions of these programs' executables using both the native `cc` compiler shipped with each of the platforms, as well the native `gcc` (version 2.5.8 on Mips, Sparc and x86, version 2.6.3 on the PowerPC). For all compilers, we use the highest available level of intra-procedural 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 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.

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	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 5: Execution time of native code generated by `gcc` relative to native code generated by `cc`

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 due to (iii) and (iv) by comparing the native `cc` and `gcc` compilers. Table 5 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 [20], especially when the compiler performs global instruction scheduling [6]. We are currently enhancing our translators to include more aggressive global instruction scheduling, and a more general framework for machine dependent peephole optimizations. We expect these improvements to bring PowerPC performance in line with the other two RISC processors. We believe that these improvements will also impact x86 performance, 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 gain over `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 impact of applying compiler optimizations to software fault isolation [36]. Based on these studies, we expect that Omniware's software fault isolation overhead will be cut to approximately 5% through optimization.

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

4.2 Benefits from translator optimizations

Table 6 shows that simple basic-block optimizations can substantially improve native code generated by OmniVM translators. This is encouraging, since mobile code applications will often require extremely fast load times, rendering infeasible the use of global optimizations. This table shows execution times of mobile code relative to the execution times of native code generated by `cc`. Comparing Table 6 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 effectively 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 6 and 3, that translator optimizations make up for some of the overhead introduced by SFI.

Even though we do not perform instruction scheduling for the Sparc, performance on the Sparc is the most competitive with the native `cc` compiler. The Sparc 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 more 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. 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:

1. 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.

2. 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.

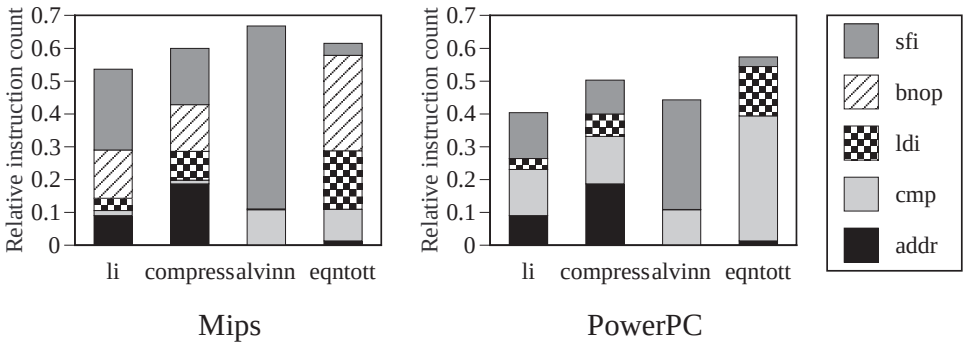

Figure 1: Expansion introduced by translation

3. 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.

4. 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.

5. 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 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 [11, 27]. For many applications of mobile code, such as executable content for Internet documents, our current performance is more than sufficient. However, we plan to aggressively add optional optimization capabilities 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 [23]. By adding these additional capabilities, we hope to make the Omniware system suitable for tasks such as general software distribution.

In addition, our data 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 Mips has a substantial number of branch nops that remain after scheduling suggesting that more aggressive branch delay slot scheduling is necessary. We also plan to take advantage of the "branch likely" instructions of the Mips R4400. Third, 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. Aggressive machine-dependent optimizations targeting these instructions is necessary on the PowerPC. Some comparisons against zero can be eliminated on the PowerPC by folding the setting of the condition codes into a prior arithmetic instruction (the Sparc can also benefit from this optimization). 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%. Despite this overhead, our performance is still good. The overhead of SFI optimizations can be reduced using standard compiler techniques such as loop invariant code motion, as described in [36]. We have not implemented SFI optimizations and expect optimization will cut this overhead in half.

5 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, and we are currently implementing these optimizations. To our knowledge, Omniware is the fastest system for mobile code to date, and the first to efficiently implement safe, mobile 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] Kristy Andrews and Duane 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] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.

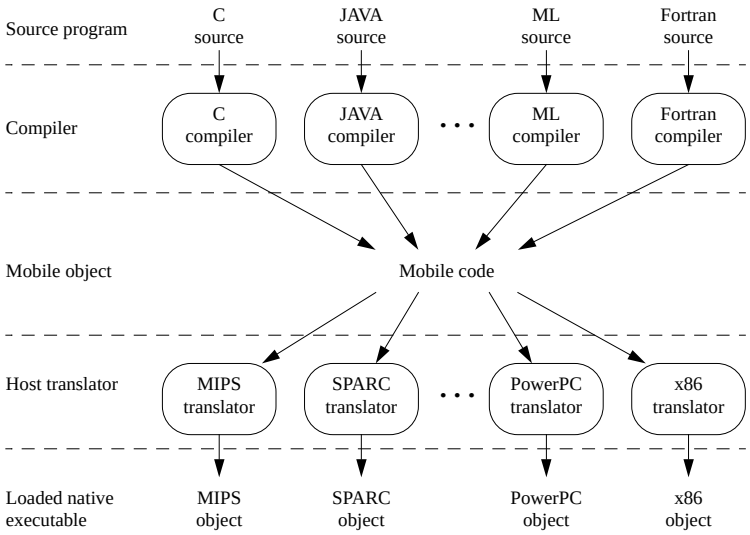

Figure 2: A universal mobile code substrate.

[3] H. E. Bal, A. S. Tanenbaum, and M. F. 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] Tim Berners-Lee, Roy T. Fielding, and Henrik Frystyk Nielsen. HTTP/1.0 Internet Draft 04, October 1995. Internet Draft (work in progress).

[6] David Bernstein and Michael 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] Nathaniel S. 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] 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.

[11] K. Chung and H Yuen. A Tiny Pascal Compiler. *Byte*, 39(9):58–64, September 1978.

[12] Compton's Interactive Encyclopedia, 1995.

[13] J Strong et al. The Problem of Programming Communication with Changing Machines. *Communications of the ACM*, 1(8):12–18, August 1958.

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

[15] C. W. Fraser and D. R. Hanson. A Retargetable Compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, Oct 1991.

[16] Gcc, 1994. Free Software Foundation.

[17] James Gosling. Java Intermediate Bytecodes. In *ACM SIGPLAN Workshop on Intermediate Representations (IR '95)*, pages 111–118, January 1995.

[18] James Gosling and Henry McGilton. The Java Language Environment: A White Paper, 1995. Sun Microsystems, Inc.

[19] Susan Graham, Steven Lucco, and Robert Wahbe. Adaptable Binary Programs. In *Proceedings of the 1995 Winter USENIX Conference*, pages 315–325, January 1995.

[20] C. Brian Hall and Kevin 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.

[21] Robert Harper and Peter 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.

[22] Scott Knaster. Magic Cap Concepts, May 1995. General Magic, Inc.

[23] Monica Lam, Edward Rothberg, and Michael 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.

[24] Stavros Macrakis. From UNCOL to ANDF: Progress in Standard Intermedia Languages, 1993. Open Software Foundation.

[25] 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.

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

[27] J. Ousterhaut. TCL: An Embeddable Command Language. In *Proceedings of the 1990 Usenix Winter Conference*, pages 22–26, January 1990.

[28] David A. Patterson. Reduced Instruction Set Computers. *Communications of the ACM*, 28(1):8–21, January 1985.

[29] Dick Pountain. Parallel Course. *Byte*, 19(7):53–60, July 1994.

[30] Bjarne Steensgaard and Eric Jul. Object and Native Code Thread Mobility Among Heterogeneous Computers. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995. (to appear).

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

[32] Richard E. Sweet. The mesa programming environment. In *Proceedings SIGPLAN Symposium on Language Issues in Programming Environments*, pages 216–229, July 1985.

[33] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. Technical report, School of Computer Science, Carnegie Mellon University, October 1995. To appear.

[34] Jeffrey Ullman. *Elements of ML programming*. Prentice Hall, 1994.

[35] Guido van Rossum. Python Tutorial, October 1995. Online at: http://www.python.org/doc/tut/tut.html.

[36] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficent Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, June 1993.

[37] Robert Wahbe, Steven Lucco, and Susan 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.

[38] David Wall. Experience with a Software-Defined Machine Architecture. *ACM Transactions on Programming Languages and Systems*, 14(3), July 1992.

[39] David W. Wall. Global Register Allocation at Link Time. In *Proceedings of the 7th SIGPLAN Symposium on Compiler Construction*, pages 264–275, June 1986.

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