# Omniware: A Universal Substrate for Web Programming

Steven Lucco,
Oliver Sharp,
Robert Wahbe

## Abstract

This paper describes Omniware, a system for producing and executing mobile code. Next generation Web applications will use mobile code to specify dynamic behavior in Web pages, implement new Web protocols and data formats, and dynamically distribute computation between servers and browsers. Like all mobile code systems, Omniware provides portability and safety. The same compiled Omniware module can be executed transparently on different machines, and a module's access to host resources can be precisely controlled.

In addition to portability and safety, Omniware has two unique features. First, Omniware is open. Omniware uses software fault isolation (SFI) to enforce safe execution of standard programming languages, enabling Web developers to leverage the vast store of existing software and programming expertise. For example, Omniware developers can use C++ to create programs for Web pages. Second, Omniware is fast. We evaluated Omniware under the Solaris 2.4 operating system on a SPARCstation 5 using eight C benchmark programs, including five programs from the C SPEC92 benchmark suite. We evaluated the performance of Omniware in two ways. First, we showed that Omniware modules can be represented compactly, reducing the space consumption compared to SunPro cc shared object files by an average of 38%. Second, we showed that Omniware modules execute at near native speeds. Including the runtime overhead necessary to ensure that Omniware modules are both portable and safe, our benchmark programs ran within 6% of native performance.

## Introduction

*Mobile code* is transforming next generation Web applications by adding interactivity to traditional Web content such as forms, text, graphics, audio, and video. Mobile code, like traditional software, is a sequence of executable instructions. Stand-alone programs such as Web browsers and servers dynamically incorporate mobile code to augment system capabilities or to enhance the presentation of a document.

Web developers are using mobile code to create a new class of dynamic Web applications. For example, Starwave plans to use mobile code to enhance their sports-related Web pages; these pages will feature a continuous ticker-tape of up-to-the-minute sports statistics. Web system developers can also use mobile code to simplify incorporation of evolving data formats and network protocols. For instance, as the Web community converges on a standard for electronic commerce, each revision of the standard billing protocol can be encapsulated as a set of mobile code modules and disseminated to Web browsers and servers. Browser users will no longer need to explicitly update their software to incorporate each new standard.

Finally, Web system developers can use mobile code to dynamically partition Web application functions between Web servers and browsers. Web systems can use dynamic application partitioning to increase an application's quality of service in the face of resource shortages, such as a lack of network bandwidth. When confronted with a slow modem connection, for example, an application can choose to regenerate graphics on the client machine rather than ship finished graphics from the server.

Consider the deployment of a 3D building walk-through service on the Web. A person relocating to a new city might use such an application to remotely tour prospective homes. Without mobile code, the application would require network messages to receive and respond to mouse and keyboard input. New screen images, perhaps in response to a moving joystick, would have to be calculated on the server and sent as data to the browser. Because all processing and data production take place at the server, each new user of the system imposes a burden on server processing resources and communication bandwidth. A Web developer can alleviate both of the difficulties in this scenario--slow interactive response time and limited server resources--by using mobile code to provide application-specific caching, data prefetching, and interactive response.

## Mobile Code Systems

To support this vision of next generation Web systems, mobile code must be portable and safe. The same mobile code module will be retrieved and executed by Web tools running under different operating systems and on top of different hardware architectures. Given the striking heterogeneity of the Internet, it is crucial that mobile code be operating system and hardware independent. Safety is also a key requirement; as users browse the Web, they will be downloading and executing hundreds of these executable code modules so the browser must be in complete control over each module's access to the host system.

In this paper we introduce Omniware, a new mobile code system. Like all mobile code systems introduced so far, Omniware delivers both portability and safety. Portability is achieved by using a virtual machine called the OmniVM. For maximum performance and power, OmniVM is modeled after an enhanced RISC processor. When OmniVM loads a mobile code module, it dynamically compiles the module into native machine code. Safety is ensured through the use of software fault isolation technology [SFI93]. Software fault isolation (SFI) inserts specialized checking code into the module's native instruction stream so that an module's access to all resources can be controlled.

Omniware has two unique features.

*First, Omniware is open.* Because Omniware uses SFI to enforce safety, it can support *any* programming language. The current Omniware system provides a C and C++ development environment called OmniC++; work on a Visual Basic front-end is in progress. We began with C and C++ for two reasons. First, support for standard languages eliminates the time-consuming process of mastering a new language. Second, OmniC++ enables developers to incorporate legacy code into their Web applications. For example, a database front-end developer, using Omniware, can port the front-end to a Web page without having to rewrite the bulk of the interface program or learn a new programming language.

More generally, Web applications, like traditional applications, will require functionality in the areas of data structures, data format conversions, security, network protocols, image processing, event handling, and text search engines. This type of software infrastructure, and much of the world's software infrastructure, is held together by a vast base of C and C++ programs and libraries. Because of this, all high-level language systems, such as Visual Basic, Tcl, and PowerScript, provide the means for developers to specify complex or time-consuming tasks in C. Most of the major office tools, such as word processors, spreadsheets, and presentation systems, are written predominantly in C or C++.

Even applications developed with a rapid development language such as Visual Basic rely on third-party programs such as VBXs or OCXs, written in C and C++, to add power and professionalism to their interfaces. Typically, such applications have over 90% of their code written in a high-level language like Visual Basic, but spend over 90% of their processing cycles in small but crucial C++ modules. OmniC++ provides a means for Web developers to apply the existing C and C++ programming infrastructure directly to Web pages.

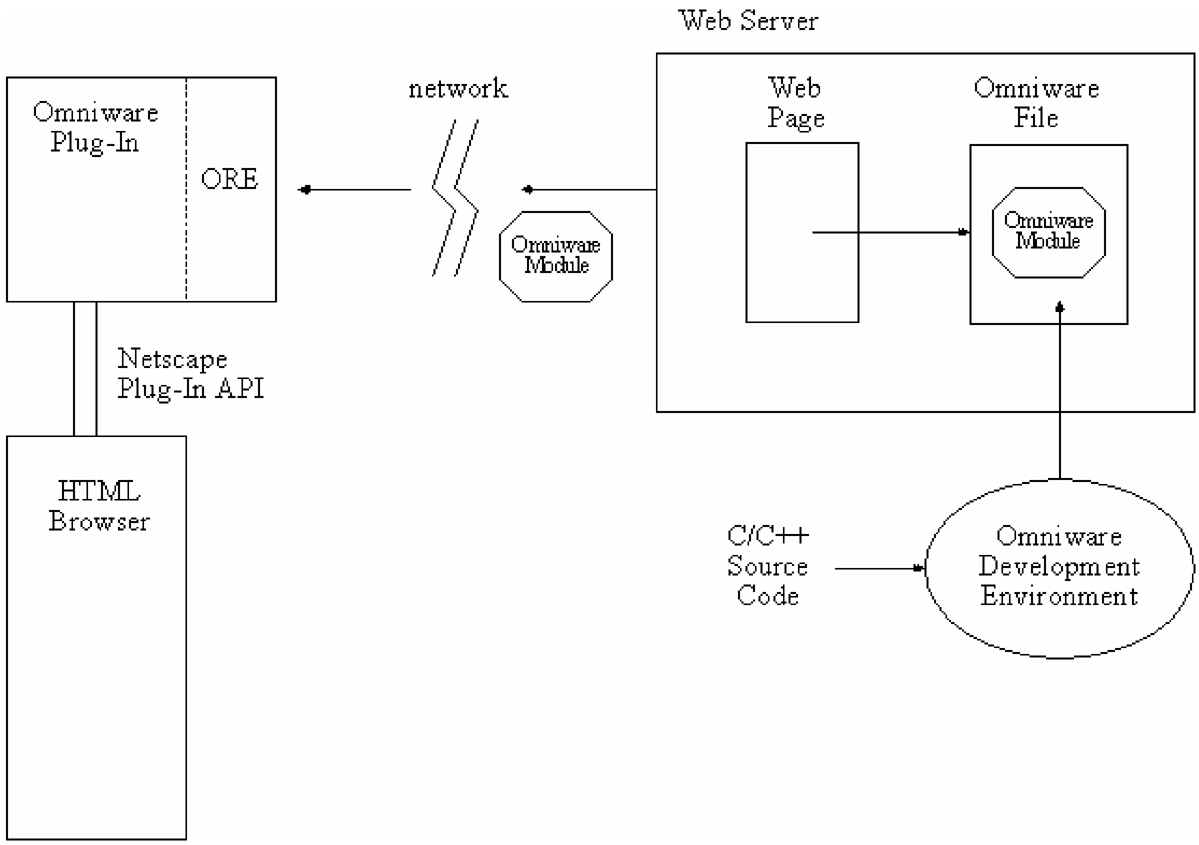
*Second, Omniware is fast.* We evaluated Omniware under the Solaris 2.4 operating system on a SPARCstation 5 using eight C benchmark programs, including five programs from the C SPEC92 benchmark suite. The performance of the system was evaluated in two ways. First, we showed that Omniware modules can be represented compactly, reducing the space consumption compared to SunPro cc shared object files by an average of 38%. Second, we showed that Omniware modules execute at near native speeds. Each benchmark program was compiled into an Omniware module. We then ran these modules as mobile code within a simple browser shell. The average runtime overhead for safe and portable Omniware modules was 6% compared to fully optimized C code. These tests were performed with a beta version of Omniware and we expect that a tuned implementation will reduce these numbers even further.

Because it supports standard languages such as C++ at near native performance, Omniware is uniquely positioned to provide a universal substrate for programming the World Wide Web. Developers can use higher level interpreted languages such as Visual Basic, Tcl, Lisp, and Perl, simply by incorporating their runtime interpreters as Omniware modules. For example, in section 4, we show that an Omniware version of the xlisp interpreter incurs only 5.1% execution overhead. Similarly, the standard Tcl interpreter run as an Omniware module incurs only 5.2% execution overhead. These interpreted languages can be mixed safely with C and C++ modules, as necessary. In short, Omniware enables Web programmers to apply to Web pages the same rapid development techniques used in creating graphical desktop application programs.

This paper is organized as follows: Section 2 provides an overview of the Omniware system. Section 3 describes, in some detail, the Omniware Runtime Environment. Section 4 presents our performance results. Section 5 discusses related work and Section 6 concludes.

# System Overview

The Omniware system enables Web browsers to manage compiled mobile code modules called *Omniware modules*. The Omniware Runtime Environment (ORE) Plug-In enables any browser that supports Netscape's Plug-In API to run Omniware. Web system programmers can use the Netscape Plug-In API to add new facilities, such as Macromedia's Director, Sun's Java, and Colusa's Omniware to existing Web browsers and servers. Several browser companies have already announced support for Netscape's Plug-In API. As other plug-in APIs become accepted, Colusa will release compatible plug-in modules. The architecture for the system is shown in the diagram below.




An Omniware-enabled browser supports execution of mobile code embedded in Web pages, just as today's browsers support GIF and JPEG images. When a module is embedded into a page, the HTML command that refers to it determines whether the module is executed automatically when the user loads the page (like an inlined image that displays immediately) or whether it acts like an external reference so that execution does not start until the user clicks on the appropriate link in the browser.

The ORE plug-in controls the execution of a module and specifies the resources that are available to it. For example, the plug-in carefully restricts a module's access to memory. Modules can allocate and use a limited amount of memory but are prevented from accessing memory that is private to the browser.

Modules interact with the rest of the system and with the user through the Omni32 API. The Omni32 API includes routines to manage memory, threads, and I/O; it also includes a graphics and windowing library that is based on the abstractions used in the existing systems Tk [Tcl94] and Java AWT [Java95] (both of which are discussed in the related work section).

The API functions check their arguments carefully, so that modules cannot use them as surrogates to violate protection safeguards. Omniware modules invoke the functions simply by calling them - there is no special linkage interface. When the module is loaded by the plug-in, it identifies any references to exported functions and routes the calls appropriately.

Programmers build mobile code using the OmniC++ development environment, which compiles C and C++ programs into Omniware modules. OmniC++ is a full and unrestricted implementation of K&R C, ANSI C, and ANSI C++. OmniC++ also includes a graphical debugger that works with Omniware modules. Colusa plans to release a Visual Basic programming environment to support rapid development of Web applications.

# Omniware Runtime Environment

This section describes the Omniware Runtime Environment (ORE). The ORE consists of two components: the Omniware Virtual Machine (OmniVM) and the Omni32 API. Compiled Omniware modules execute on top of OmniVM just as conventional programs execute on standard hardware. The Omni32 API provides services typically handled by either conventional operating systems or standard system libraries.

Just as it is not necessary to understand the system architecture of a conventional desktop computer in order to develop programs for it, it is not necessary to understand OmniVM to develop Omniware modules. The discussion of OmniVM that follows is provided for readers who are interested in the lowest-level details of the system. It explains how OmniVM provides portability and near-native performance while maintaining strict control over the execution of Omniware modules. These details are invisible to programmers who write modules in high-level languages.

## Omniware Virtual Machine

The design of OmniVM reflects four goals: safety, mobility, performance, and openness. To make Omniware an open system, we designed OmniVM to be a straightforward compilation target for a large variety of source languages. To achieve excellent performance, we used instruction mix and memory system traces from existing RISC and CISC processors to determine the instruction set for OmniVM. To support mobility, we standardized OmniVM floating point and integer data formats and introduced OmniVM instructions that are compatible with the data formats of existing processors. Finally, to ensure safe execution of Omniware modules, we designed the OmniVM instruction set to support straightforward implementation of software fault isolation.

### OmniVM Instruction Set Architecture

OmniVM is a RISC processor enhanced in several respects with high-level (CISC-like) features. The high-level enhancements to OmniVM support portability and performance. For example, the OmniVM instruction set includes a memory-to-memory block move instruction (`mov.b`) analogous to the `rep movsb` instruction on the Intel Pentium processor. This instruction gives the OmniVM translator the opportunity to generate optimal block move code for the target machine. The Pentium OmniVM translator converts `mov.b` directly to `rep movsb`.

High-level instructions like `mov.b` give OmniVM translators the power to generate optimal code for the intended high-level operation. However, there are two reasons to avoid designing a virtual machine like OmniVM entirely around high-level instructions. First, the indiscriminate use of high-level instructions yields an unattractive compilation target. Because high-level instructions are specialized to a particular task, they require a compiler to identify and handle more distinct code generation cases. Similarly, the use of high-level instructions increases the complexity of each OmniVM translator.

But there is a more fundamental reason to prefer a simple, orthogonal design for the backbone of the OmniVM instruction set. A high-level language compiler can not do a good job of OmniVM instruction selection because it has no information about the relative timing of OmniVM instructions. For example, the OmniVM is a load/store architecture. Suppose that we chose instead to provide memory addressed operands to instructions such as the add signed integer instruction (`add.iw`). Without instruction timing information, a compiler might select to read an operand from memory, rather than use an extra instruction to regenerate that operand. On the Intel Pentium, this is sometimes a good decision. On most RISC architectures, regenerating an integer value with one or two register instructions will generally outperform reloading that value from memory.

To summarize, these were the design rules we followed for the OmniVM instruction set:

- If a high-level instruction can be synthesized from RISC instructions, choose the RISC instructions.
- If a high-level instruction (such as block move) can not be synthesized from simpler instructions, include the high-level instruction if instruction traces of representative applications on CISC machines (including the Motorola 68000 and Intel Pentium) include significant use of the high-level instruction.
- Choose a high-level instruction if it is necessary to standardize the handling of integer and floating point data formats.

The latter point is motivated by the central design goal for Omniware: if an Omniware module runs correctly on one processor architecture, it should run correctly on all other processor architectures without modification. This guarantee enormously simplifies the implementation of documents containing mobile code.

**OmniVM Protection Architecture**

To enforce protection constraints on Omniware modules, the OmniVM provides a simple protection architecture. The OmniVM protection architecture resembles in function the memory management unit (MMU) of a single computer. OmniVM supports a segmented virtual address space - it divides addressable memory into segments, the size of which is fixed for a given instance of the OmniVM. Supervisor can use the OmniVM protection architecture to set the segment size and to define memory contexts called *protection domains*. Each protection domain has its own segment table that specifies its memory access permissions.

The ORE exports this functionality to trusted code such as browser software through the following interface:

```
int omni_get_segment_size()

OmniBoolean omni_mprotect(OmniPd *pd, void *addr,
int size, OmniMemPerm mode)
```

The function `omni_mprotect` changes the access permissions on the mappings specified by the range `<addr, addr+size>` to be those specified by `mode`. `addr` and `size` must be multiples of the segment size as returned by the function `omni_get_segment_size()`. Read and write permission can be separately specified. Using this interface, a host program can specify precisely what data a protection domain is allowed to read and write.

When the ORE loads an Omniware module, it creates a new protection domain for the module. By default, a thread in the new protection domain can only access its stack and the code and data of the

Omniware module associated with the protection domain. A host program such as a browser can direct the ORE to load multiple Omniware modules into a single protection domain; the modules would then share memory access permissions.

The ORE uses a technology called software fault isolation (SFI) to implement the semantics of the OmniVM protection architecture. This technology is described extensively elsewhere [SFI93]. Three essential properties of software fault isolation are:

- It is based on the semantics of the underlying processor architecture and not the high-level source language.
- It uses a form of runtime checking that can be heavily optimized so that protection overhead is small (see performance measurements in [SFI93] and section 4 of this paper).
- A software system can separate the *verification* of a fault isolated module from its production. Verification is a simple, linear-time procedure.

### Omni32 API

The Omniware Runtime Environment contains operating system independent libraries for graphics and windowing, memory management, file I/O, threads, synchronization and signals. Functions that may be called by an untrusted module verify the validity of their arguments. For example, the memory management routine to deallocate memory, `free`, gracefully handles being passed an invalid pointer. The Omni32 API provides a comprehensive set of services for mobile code applications.

# Performance

Using eight standard benchmark programs, we evaluated the performance of Omniware under the Solaris 2.4 operating system on a SPARCstation 5 with 96 megabytes of memory. Each benchmark program was compiled, using OmniC, into an Omniware module. Each module was loaded into a browser shell and then executed as a mobile code module. The module was invoked simply by calling `main` with the appropriate arguments. Rather than modify the benchmark codes in any way, the browser shell provided a safe compatibility library for the various system calls needed by the modules. For example, when the module attempts to call the `open` system call, the Omniware Runtime Environment transparently redirects the call to `omniware_safe_open`, which then verifies its arguments.

We measured the performance of Omniware in two ways. First, we measured the size of each compiled Omniware module. Second, we measured its end-to-end execution time. For comparison, we performed these same measurements by compiling the benchmark programs into Solaris shared object files. The programs were compiled using the Solaris ANSI SunPro cc compiler and linked with the standard Solaris dynamic linking facility. All modules were compiled with the highest level of optimization. The comparison of Omniware modules to standard shared object files is done only to illustrate Omniware's performance: unlike Omniware modules, shared object files are neither safe nor portable. The results are presented in Table 1.

We chose a diverse set of standard and well-known benchmark programs. The programs `alvinn`, `ear`, `compress`, `xlisp`, and `gcc`, are part of the C SPEC92 benchmark suite. The other benchmarks are well-known and widely used programs. This diverse set of benchmarks, which includes both floating point and integer intensive codes, should help readers predict how Omniware will perform in different contexts. For the SPEC92 benchmark programs, the input to each program was the standard input used in the SPEC92 benchmark. For Tcl, it was the test suite that comes with the standard distribution. For TeX, it was a 4 page document containing complicated mathematical equations. For `lcc`, it was the `lcc` file `x86.c`.

| | | File Size (bytes) | | | Execution Time (seconds) | | |
|---|---|---|---|---|---|---|---|
| Program | Source Lines | Shared Objects | Omniware | Over-head | Shared Objects | Omniware | Over-head |
| alvinn | 1K | 17K | 12K | -42% | 307.0 | 337.0 | 6.7% |
| ear | 5K | 208K | 140K | -49% | 1014.0 | 1067.0 | 5.2% |
| compress | 2K | 11K | 7K | -57% | 5.6 | 5.5 | 2.3% |
| lcc | 25K | 194K | 163K | -19% | 4.4 | 5.2 | 7.7% |
| gcc | 325K | 839K | 687K | -22% | 8.8 | 10.9 | 9.1% |
| tcl | 27K | 106K | 97K | -9% | 23.3 | 24.5 | 5.2% |
| tex | 23K | 161K | 112K | -44% | 4.1 | 4.3 | 4.9% |
| xlisp | 8K | 59K | 37K | -59% | 53.3 | 56.0 | 5.1% |
| **Average** | | | | **-38%** | | | **5.8%** |

Table 1: Omniware performance results compared to using fully optimized native shared object files.

The average overhead for our benchmark programs was 5.8%. There are two reasons for this overhead. First, because OmniC generates machine-independent code, the output cannot be perfectly tuned to any particular architecture. While the runtime environment's dynamic compiler can attempt to perform various machine-dependent optimizations, this two phase process inherently introduces some inefficiencies. The second source of overhead is the set of runtime checks needed to control a module's access to host resources.

Omniware's low execution overhead permits developers unparalleled flexibility. For example, a new Internet programming language could be widely introduced by simply providing the runtime environment as an Omniware module. When a Web tool retrieved a high-level mobile code module in an unknown language, it could simply download the appropriate runtime interpreter. Because the runtime environments for Tcl, xlisp, and Perl are written in C, an Omniware enabled tool would automatically support these popular languages. Further, Omniware's performance enables potentially compute intensive data conversion procedures, decompression algorithms, and other extensions to be dynamically distributed as Omniware modules. In general, because Omniware executes at near native speeds, the software techniques and practices that are viable for stand-alone applications will also be viable for next generation Web applications.

# Related Work

The question of how best to provide mobile code on the Internet is attracting a great deal of attention. A number of different strategies for providing mobile code have been proposed. All of the systems described in this section are *closed* in that they cannot effectively support multiple source languages.

Sun Microsystem's Java system has three components: a new high-level object-oriented language called Java, a low-level virtual machine called the Java VM, and operating system independent libraries for file I/O, memory management, threads, synchronization, and graphic operations. Sun provides a browser, called HotJava, which supports Java programs embedded in Web pages. To execute a Java program, it is compiled into Java VM instructions that are then loaded and interpreted by the Java VM. The Java VM lacks the necessary primitives to support standard programming languages.

Current implementations of the Java VM use a bytecode interpreter. No comprehensive performance numbers have been published, but current estimates are that Java programs run 1200% to 3000% percent slower than native code. At some point in the future the Java project plans to offer a virtual machine that employs dynamic compilation. Certain features of the virtual machine, such as array bounds checking, stack based operations, and garbage collection will make it difficult to implement efficiently [GC84, GC93, Bounds92, Interp77].

Java and Omniware are similar in that both offer safety and portability. Java achieves these properties through restricting the programming language. Omniware uses software fault isolation to enforce safety, which enables Omniware to efficiently support standard programming languages.

Another approach to supporting mobile code is the Guile [Guile95] project from Cygnus Support. Its implementation is still underway at the time this paper is being written, but the idea is to provide a library that includes an interpreter for a language based on Scheme [Scheme91]. The library provides a set of data structures and system services. Like Omniware, the Guile library is linked into a host application and allows it to manage code modules. The host can declare functions that become new primitives in the Guile language and are thus available to the modules. No performance information has yet been published for Guile.

The Telescript [Telescript93] system from General Magic focuses on the development of *network agents*. These are autonomous programs that can move through a network, interacting with the hosts that execute them and with other agents that they encounter. Like Java, there are two levels of Telescript: High Telescript and Low Telescript. We have not been able to obtain detailed technical information or performance measurements for Telescript.

Safe-Python is a modified version of Python [Python94], an interpreted object-oriented scripting language that is popular for rapid development. Safe-Python is an altered version of the language that controls access to the operating system and provides additional primitives for building distributed applications. The intent is to merge the two versions of the language in the future so that Python can be used to build distributed applications and to support mobile code.

Safe-Tcl [Safe-Tcl94] is a modified version of the Tcl language [Tcl94]. Tcl was primarily designed to allow programmers to write graphical applications more quickly. Safe-Tcl is a restricted version of the full language that is safe to incorporate into a mail message. Safe-Tcl is intended to serve as an extension to the MIME mail message format that adds support for mobile code. When a message includes mobile code, MIME will send it to the safe-Tcl interpreter. Since an incoming mail message could be sent by anyone, the language needed to be carefully constrained to prevent mischief. Some of the normal Tcl language primitives are removed, some additional ones added to support integration with MIME, and a few commands are slightly altered.

# Conclusion

A universal substrate for Web programming must be fast, safe, portable, and open. Speed is required so that no artificial restrictions are placed on mobile code and to allow sophisticated runtime systems, such as Visual Basic interpreters, to be implemented as mobile code modules. Safety is important so that users and developers need only trust the mobile code substrate and not the myriad higher-level applications and libraries that might be employed. Given the striking heterogeneity of the Internet, portability is crucial. If a mobile code module runs correctly on one processor architecture, it should run correctly on all other processor architectures without modification. Finally, a Web programming substrate should enable developers to apply to Web pages the vast existing base of desktop programming infrastructure.

This paper described a fast, safe, portable, and open system for Web development called Omniware. The Omniware virtual machine executes Omniware modules at native speeds. The average overhead among our eight benchmark programs was 6%. Through the use of software fault isolation technology, the host application can precisely control a module's access to resources. The same compiled Omniware module can execute, without modification, across heterogeneous operating system and hardware architectures. Because Omniware does not rely on language semantics to enforce safety, it does not force Web programmers to remain within the confines of a single, non-standard programming language. Omniware frees programmers to choose the combination of high and low-level programming language techniques most appropriate to a given development task, and delivers along with the ability to use standard programming techniques the performance that desktop developers expect.

# References

[Bounds92] J. L. Stefen, "Adding Run-Time Checking to the Portable C Compiler," Software - Practice and Experience, April 1992, vol.22, no.4, p. 305-16.

[GC84] R. Brooks, "Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware," ACM Software Engineering Symposium on Practical Software Development Environments, 1984, p. 256-262.

[GC93] B. Zorn, "The Measured Cost of Conservative Garbage Collection," Software - Practice and Experience, vol 23, no. 7, July 1993, p. 733-56.

[Guile95] T. Lord, "The Guile Architecture for Ubiquitous Computing," *to appear in:* Usenix Tcl/Tk Workshop, 1995.

[HTML95] I. S. Graham. *The HTML Sourcebook*, Wiley: New York, 1995.

[IEEE85] *IEEE Standard 754-1985. IEEE Standard for Binary Floating-Point Arithmetic.* IEEE: New York, 1985.

[Interp77] J. P. Fitch and A. C. Norman, "Implementing LISP in a High-Level Language," Software - Practice and Experience, vol. 7, 1977, p. 713-725.

[Java95] J. Gosling. "Java Intermediate Bytecodes," ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, Jan. 1995.

[Pentium94] *Pentium Processor User's Manual*. Intel Corporation: Mt. Prospect, IL, 1994.

[Perl92] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly and Associates: Sebastopol, CA, 1992.

[Python94] http://minsky.med.virginia.edu/sdm7g/Projects/Python/SafePython.html

[Safe-Tcl94] N. S. Borenstein. "Email With a Mind of its Own: The Safe-Tcl Language for Enabled Mail," IFIP International Conference, Barcelona, Spain, June 1994.

[Scheme91] J. Rees and W. Clinger, eds. "The Revised^4 Report on the Algorithmic Language Scheme," ACM Lisp Pointers, vol. 4, no. 3, 1991.

[SFI93] R. Wahbe, S. Lucco, T. Anderson, and S. L. Graham. "Efficient Software-Based Fault Isolation," 14th ACM Symposium on Operating Systems Principles, Ashville, NC, Dec. 1993.

[SPEC92] *SPEC92 Release Notes*. Standard Performance Evaluation Corporation (SPEC): Fairfax, VA, 1992.

[Tcl94] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley: Reading, Mass., 1994.

[Telescript93] "Telescript Technology: The Foundation for the Electronic Marketplace," General Magic: Sunnyvale, CA, 1993.

[WWW94] T. Berners-Lee, R. Cailliau, A. Loutonen, H. F. Nielsen and A. Secret. "The World-Wide Web," Communications of the ACM, vol. 37, no. 8, August 1994, p. 76-82.