17th ACM Symposium on Operating System Principles (SOSP'99)
Published as *Operating Systems Review* **34**(5):202–216, Dec. 1999

Design and implementation of a distributed virtual machine for networked computers

Emin Gün Sirer, Robert Grimm, Arthur J. Gregory, Brian N. Bershad

University of Washington
Department of Computer Science and Engineering

{egs, rgrimm, artjg, bershad}@cs.washington.edu

Abstract

This paper describes the motivation, architecture and performance of a distributed virtual machine (DVM) for networked computers. DVMs rely on a distributed service architecture to meet the manageability, security and uniformity requirements of large, heterogeneous clusters of networked computers. In a DVM, system services, such as verification, security enforcement, compilation and optimization, are factored out of clients and located on powerful network servers. This partitioning of system functionality reduces resource requirements on network clients, improves site security through physical isolation and increases the manageability of a large and heterogeneous network without sacrificing performance. Our DVM implements the Java virtual machine, runs on x86 and DEC Alpha processors and supports existing Java-enabled clients.

1. Introduction

Virtual machines (VMs) have the potential to play an important role in tomorrow's networked computing environments. Current trends indicate that future networks will likely be characterized by mobile code [Thorn 97], large numbers of networked hosts per domain [ISC 99] and large numbers of devices per user that span different hardware architectures and operating systems [Hennessy 99, Weiser 93]. A new class of virtual machines, exemplified by systems such as Java and Inferno [Lindholm & Yellin 96, Dorward et al. 97], has recently emerged to meet the needs of such an environment. These modern virtual machines are compelling because they provide a platform-independent binary format, a strong type-safety guarantee that facilitates the safe execution of untrusted code and an extensive set of programming interfaces that subsume those of a general-purpose operating system. The ability to dynamically load and safely execute untrusted code has already made the Java virtual machine a ubiquitous component in extensible systems ranging from web browsers and servers to database engines and office applications. The platform independence of modern virtual machines makes it feasible to run the same applications on a wide range of computing devices, including embedded systems, handheld organizers, conventional desktop platforms and high-end enterprise servers. In addition, a single execution platform offers the potential for unified management services, thereby enabling a small staff of system administrators to effectively administer thousands or even hundreds of thousands of devices.

While modern virtual machines offer a promising future, the present is somewhat grim. For example, the Java virtual machine, despite its commercial success and ubiquity, exhibits major shortcomings. First, even though the Java virtual machine was explicitly designed for handheld devices and embedded systems, it has not been widely adopted in this domain due to its excessive processing and memory requirements [Webb 99]. Second, it is the exception, rather than the rule, to find a secure and reliable Java virtual machine [Dean et al. 97]. And third, rather than simplifying system administration, modern virtual machines, like Java, have created a substantial management problem [McGraw & Felten 96], leading many organizations to simply ban virtual machines altogether [CERT 96].

We assert that these symptoms are the result of a much larger problem that is inherent in the design of modern virtual machines. Specifically, state of the art modern virtual machines rely on the monolithic architecture of their ancestors [Goldberg 73, Popek & Goldberg 74, IBMVM 86, UCI 96]. All service components in a monolithic VM, such as verification, security management, compilation and optimization, reside locally on the host intended to run the

SOSP-17 12/1999 Kiawah Island, SC

VM applications. Such a monolithic service architecture exhibits four shortcomings:

1. Manageability: Since each modern virtual machine is a completely independent entity, there is no central point of control in an organization. Transparent and comprehensive methods for distributing security upgrades, capturing audit trails and pruning a network of rogue applications are difficult to implement.
2. Performance: Modern virtual machine services, such as authentication, just-in-time compilation and verification, have substantial processing and memory requirements. Consequently, monolithic systems are not suitable for hosts, such as embedded devices, which lack the resources to support a complete virtual machine.
3. Security: The trusted computing base (TCB) of modern VMs is not small, well-defined, or physically isolated from application code. A large TCB with ill-defined boundaries makes it difficult to construct and certify secure systems [Saltzer & Shroeder 75]. The lack of separation between virtual machine components means that a flaw in any component of the virtual machine can place the entire machine at risk [McGraw & Felten 99]. Further, co-location of VM services has resulted in non-modular systems that can exhibit complex inter-component interactions, as observed for monolithic operating systems [Accetta et al. 86, Bershad et al. 95, Engler et al. 95].
4. Scalability: Monolithic virtual machines are difficult to port across the diverse architectures and platforms found in a typical network [Seltzer 98]. In addition, they have had problems scaling over the different usage requirements encountered in organizations [Rayside et al. 98].

The goal of our research is to develop a virtual machine system that addresses the manageability, performance, security and scalability requirements of networked computing. In addition, such a system should preserve compatibility with the wide base of existing monolithic virtual machines in order to facilitate deployment. To this end, we focus on implementation techniques that preserve the external interfaces [Lindholm & Yellin 96] and platform APIs [Gosling & Yellin 96] of existing virtual machines.

We address the problems of monolithic virtual machines with a novel distributed virtual machine architecture based on service factoring and distribution. A distributed service architecture factors virtual machine services into logical components, moves these services out of clients and distributes them throughout the network. We have designed and implemented a distributed virtual machine for Java based on this architecture. Our DVM includes a Java runtime, a verifier, an optimizer, a performance monitoring service and a security manager. It differs from existing systems in that these services are factored into well-defined components and centralized where necessary.

The rest of the paper is structured as follows. The next section describes our architecture and provides an overview of our system. Section 3 describes the implementation of conventional virtual machine services under our architecture. Section 4 presents an evaluation of the architecture and Section 5 shows how a new optimization service can be accommodated under this architecture. Section 6 discusses related work; Section 7 concludes.

2. Architecture overview

The principal insight behind our work is that centralized services simplify service management by reducing the number and geographic distribution of the interfaces that must be accessed in order to manage the services. As illustrated by the widespread deployment of firewalls in the last decade [Mogul 89, Cheswick & Bellowin 94], it is far easier to manage a single, well-placed host in the network than to manage every client. Analogously, we break monolithic virtual machines up into their logical service components and factor these components out of clients into network servers.

The service architecture for a virtual machine determines *where*, *when* and *how* services are performed. The location (i.e. where), the invocation time (i.e. when), and the implementation (i.e. how) of services are constrained by the manageability, integrity and performance requirements of the overall system, and intrinsically involve engineering tradeoffs. Monolithic virtual machines represent a particular design point where all services are located on the clients and most service functionality, including on the fly compilation and security checking, is performed during the run-time of applications. While this paper shows the advantages of locating services within the network, changing the location of services without regard for their implementation can significantly decrease performance as well. For instance, a simple approach to service distribution, where services are decomposed along existing interfaces and moved, intact, to remote hosts, is likely to be prohibitively expensive due to the cost of remote communication over potentially slow links and the frequency of inter-component interactions in monolithic virtual machines. We describe an alternative design where service functionality is factored out of clients by partitioning services into static and dynamic components and present an implementation strategy that achieves performance comparable to monolithic virtual machines.

In our distributed virtual machine, services reside on centralized servers and perform most of their functionality statically, before the application is executed. Static service

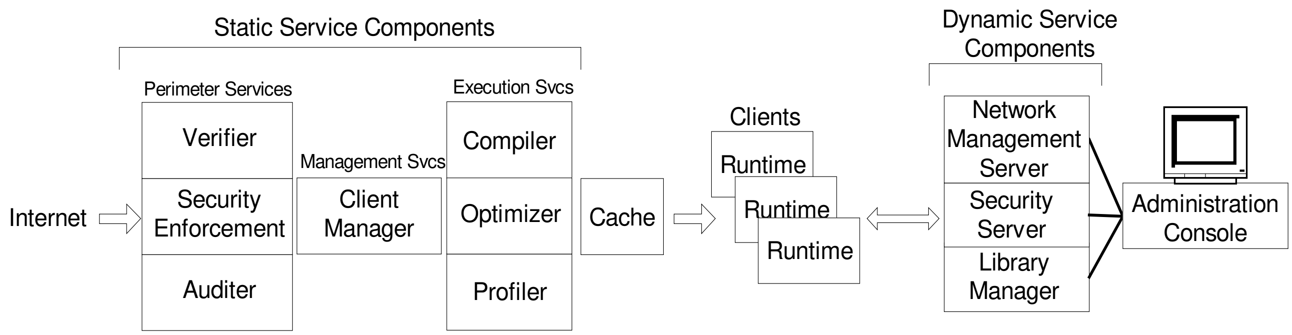

Figure 1. The organization of static and dynamic service components in a distributed virtual machine.

components, such as a verifier, compiler, auditor, profiler, and optimizer, examine the instruction segment of applications prior to execution to ensure that the application exhibits the desired service properties. For example, a verifier may check the code for type-safety, a security service may examine the statically determinable arguments to system calls, and an optimizer may check code structure for good performance along a particular path.

The dynamic service components provide service functionality during the execution of applications. They complement static service components by providing the services that inherently need to be executed at application run-time in the context of a specific client. For example, a security service may check user-supplied arguments to system calls, a profiler may collect run time statistics, and an auditing service may generate audit events based on the execution of the application.

The glue that ties the static and dynamic service components together is binary rewriting. When static service components encounter data-dependent operations that cannot be performed statically, they insert calls to the corresponding dynamic service components. For example, our static verification service checks applications for conformance against the Java VM specification. Where static checking cannot completely ascertain the safety of the program, the static verifier modifies the application so that it performs the requisite checks during its execution. The resulting application is consequently self-verifying because the checks embedded by the static service component are an integral part of the application code.

Figure 1 illustrates our distributed virtual machine architecture. Static service components produce self-servicing applications, which require minimal functionality on the clients. Dynamic service components provide service functionality to clients during run-time as necessary. The static services in our architecture are arranged in a virtual pipeline that operates on application code, as shown in Figure 2.

A distributed service architecture allows the bulk of VM service functionality to be placed where it is most convenient. A natural service placement strategy is to structure the static service components as a transparent network proxy, running on a physically secure host. Placed at a network trust boundary, like a firewall, such a proxy can transparently perform code transformations on *all* code that is introduced into an organization. In some environments, the integrity of the transformed applications cannot be guaranteed between the server and the clients, or users may introduce code into the network that has not been processed by the static services. In such environments, digital signatures attached by the static service components can ensure that the checks are inseparable from applications [Rivest et al. 78, Rivest 92], and clients can be instructed to redirect incorrectly signed or unsigned code to the

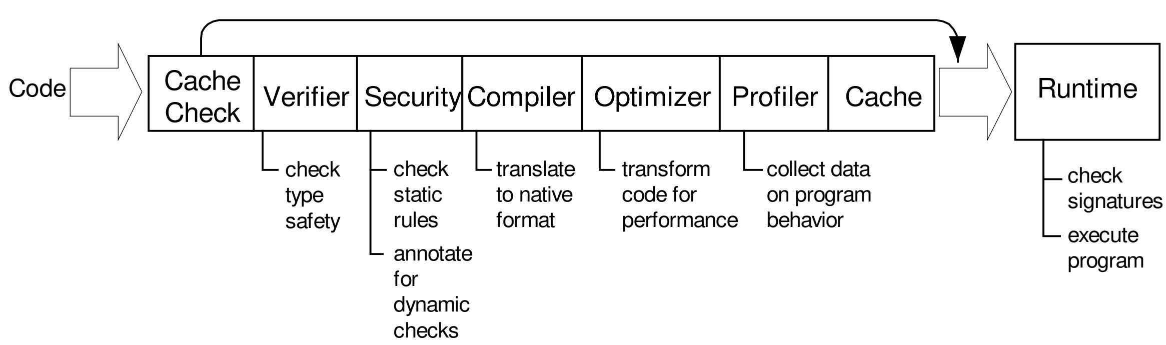

Figure 2. The flow of code through a pipeline of static service components in a distributed virtual machine. The ordering of services in this pipeline may be modified to suit organizational or functional requirements. Further, the client runtime may communicate with the static service components for client-specific services.

centralized services [Spyglass 98].

A DVM introduces a modest amount of new functionality into the existing trusted computing base of an organization. A DVM client needs to trust that the static and dynamic service components it relies on for safety, including the proxy and binary rewriter, are implemented correctly. In addition, any service authentication scheme used in the clients, which may include a digital signature checker and a key manager, form part of the trusted computing base under our design. However, we believe the actual impact of these additions to the TCB to be small. Monolithic clients already trust all of the service components that form a traditional VM and often already have provisions for cryptographic protocols and digital signature checking to support their target applications [Gong 99]. Overall, a modest increase in the TCB enables DVM clients to migrate the trusted components to physically secure, professionally managed and administered hosts, which is critical to addressing the operational problems that have plagued monolithic VMs.

Our service architecture is unique in several fundamental ways. First, the centralized services are mandatory for all clients in an organization. For example, security checks injected into incoming code are inseparable from applications at the time of their execution and are thus binding throughout the network. Second, there is a single logical point of control for all virtual machines within an organization. In the case of the security service, policies are specified and controlled from a single location; consequently, policy changes do not require the cooperation of unprivileged users. And third, using binary rewriting as a service implementation mechanism preserves compatibility with existing monolithic virtual machines. A monolithic virtual machine may subject the rewritten code to redundant checks or services, but it can take advantage of the added functionality without any modifications.

While a distributed service architecture addresses the problems faced by monolithic virtual machines, it may also pose new challenges. Centralization can lead to a bottleneck in performance or result in a single point of failure within the network. These problems can be addressed by replicated or recoverable server implementations. The next section shows how the separation between static and dynamic service components can be used to delegate state-requiring functionality to clients. Section 4 shows that this implementation strategy does not pose a bottleneck for medium sized networks even in the worst possible case and can easily be replicated to accommodate large numbers of hosts.

3. Services

We have implemented the architecture described in the previous section to support a network of Java virtual machines (JVMs). In this section, we describe the implementation of conventional virtual machine services under our architecture and show that the distributed implementation of these services addresses the shortcomings of monolithic VMs outlined in the first section. Our services are derived from the Java VM specification, which broadly defines a type-safe, object-based execution environment. Typical implementations consist of a verifier, which checks object code for type-safety, an interpreter and a set of runtime libraries. In some implementations, the interpreter is augmented with a just-in-time compiler to improve performance. The following sections describe the design and implementation of the services we have built to supplant those found in traditional Java virtual machines.

All of our services rely on a common proxy infrastructure that houses the static service components. The proxy transparently intercepts code requests from clients, parses JVM bytecodes and generates the instrumented program in the appropriate binary format. An internal filtering API allows the logically separate services described in this section to be composed on the proxy host. Parsing and code generation are performed only once for all static services, while structuring the services as independent code-transformation filters enables them to be stacked according to site-specific requirements [Heidemann & Popek 94, O'Malley & Peterson 92]. The proxy uses a cache to avoid rewriting code shared between clients and generates an audit trail for the remote administration console. The code for the dynamic service components resides on the central proxy and is distributed to clients on demand.

While the implementation details of our virtual machine services differ significantly, there are three common themes among all of them:

- Location: Factoring VM services out of clients and locating them on servers improves manageability by reducing replicated state, aids integrity by isolating services from potentially malicious code and simplifies service development and deployment.
- Service Structure: Partitioning services into static and dynamic components can enhance performance by amortizing the costly parts of a service across all hosts in the local network.
- Implementation Technique: Binary rewriting is used to implement services transparently. Binary rewriting services can be designed to incur a relatively small performance overhead while retaining backward-compatibility with existing clients.

3.1 Verification

A comprehensive set of safety constraints allows a virtual machine to integrate potentially malicious code into a privileged base system [Stata & Abadi 98, Freund &

Mitchell 98]. Indeed, Java's appeal for network computing stems principally from its strong safety guarantees, which are enforced by the Java verifier.

The task of verifying Java bytecode has been a challenge for monolithic virtual machines. First, since the Java specification is not formal in its description of the safety axioms, there are differences between verifier implementations. Verifiers from different vendors differ on underspecified issues such as constraints on the uses of uninitialized objects, subroutine calls, and cross-validation of redundant data in class files. Second, monolithic implementations tie the verifier to the rest of the VM, thereby prohibiting users from using stronger verifiers where necessary. Furthermore, monolithic verifiers make it difficult to propagate security patches to all deployed clients in a timely manner. As a case in point, 15% of all accesses to our web site originate from out-of-date browsers with well-known security holes for which many patches have been issued. Finally, the memory and processing requirements of verification render monolithic VMs unsuitable for resource limited clients, such as smart cards and embedded hosts [Cohen 97]. Some monolithic virtual machines for embedded and resource-limited systems have abandoned verification altogether for a restricted extension model based on trust [HP 99].

We address these shortcomings by decoupling verification from the rest of the VM, migrating its functionality out of clients into a network verification service and centralizing the administration of this service. Moving verification out of clients poses some challenges, however, because parts of the verification process require access to client namespaces and have traditionally required close coupling with the client JVM. Specifically, Java verification consists of four separate phases. The first three operate on a single class file in isolation, respectively making sure that the class file is internally consistent, that the code in the class file respects instruction integrity and that the code is type-safe. The fourth phase checks the interfaces that a class imports against the exported type signatures in its namespace, making sure that the assumptions that the class makes about other classes hold during linking.

In our implementation, the first three phases of verification are performed statically in a network server, while the link-time checks are performed by a small dynamic component on the client. This partitioning of functionality eliminates unnecessary communication and simplifies service implementation. During the processing of the first three phases, the verification service collects all of the assumptions that a class makes about its environment and computes the scope of these assumptions. For example, fundamental assumptions, such as inheritance relationships, affect the validity of the entire class, whereas a field reference affects only the instructions that rely on the reference. Having determined these assumptions and their scope, the verification service modifies the code to perform the corresponding checks at runtime by invoking a simple service component (Figure 3). Since most safety axioms have been checked by this time, the functionality in the dynamic component is limited to a descriptor lookup and string comparison. This lazy scheme for deferring link phase checks ensures that the classes that make up an application are not fetched from a remote, potentially slow, server unless they are required for execution.

The distributed verification service propagates any errors to the client by forwarding a replacement class that raises a verification exception during its initialization. Hence, verification errors are reflected to clients through the regular Java exception mechanisms. Since the Java VM specification intentionally leaves the time and manner of verification undefined except to say that the checks should be performed before any affected code is executed, our

```
class Hello {
    static boolean __mainChecked = false; // Inserted by the verifier
    public static void main() {
        if(__mainChecked == false) {  // Begin automatically generated code
            RTVerifier.CheckField("java.lang.System","out",
                                  "java.io.OutputStream");
            RTVerifier.CheckMethod("java.io.OutputStream","println",
                                  "(Ljava/lang/String)V");
            __mainChecked = true;
        }                           // End automatically generated code
        System.out.println("hello world");
    }
}
```

Figure 3. The hello world example after it has been processed by our distributed verification service. The vast majority of safety axioms are checked statically. Remaining checks are deferred to execution time, as shown in italics. The first check ensures that the `System` class exports a field named "`out`" of type `OutputStream`, and the second check verifies that the class `OutputStream` implements a method, "`println`," to print a string. The rewriting occurs at the bytecode level, though the example shows equivalent Java source code for clarity.

approach conforms to the specification.

While this approach to verification does not make the central task of verification any easier, it addresses the operational problems that have plagued monolithic clients. First, it allows a network of VMs to share the same verifier, thereby ensuring that code accepted for execution within that administrative domain is at least as strong as the constraints imposed by the central verifier. Transparent binary rewriting ensures that even existing monolithic VMs benefit from the assurance provided by the central verifier, though they will subject the code to redundant local verification. Second, an isolated verification service with clear interfaces is easier to check for correctness using automatic testing techniques [Sirer & Bershad 99]. Third, distributed virtual machine clients can be made smaller and cheaper because they do not have to support a verifier. Finally, in response to security flaws, only one software component needs to be updated per administrative domain, in contrast to having to patch every single network client, which often requires the assistance of users.

3.2 Security

The overall aim of any security service is to enforce a specific security policy across hosts within an administrative domain. Such a service should meet the following three goals to be comprehensive, easy to manage and versatile. First, it should uniformly enforce an organization's security policy across all nodes. Second, it should provide a single point of control for specifying organization-wide policies. And, third, it should allow an administrator to impose checks in any code deemed important for security.

Our architecture satisfies these goals by factoring most functionality that is critical for security out of the individual nodes into a centralized network security service. The security service forces applications to comply with an organization's security policy by inserting appropriate access checks through binary rewriting. The secured applications then execute on the individual hosts where a small enforcement manager [Grimm & Bershad 99] performs the inserted access checks in accordance with the centralized policy (Figure 4).

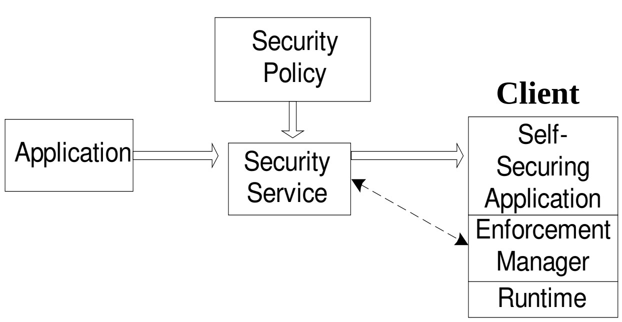

Figure 4. The structure of the security service. A master security policy determines how applications are rewritten. An enforcement manager, residing on the clients, resolves access control checks against the central policy.

Our security model derives from DTOS [Minear 95, Olawsky et al. 96, SCC 97], where security identifiers, representing protection domains, are associated with threads and security-critical objects. Permissions represent the right to perform an operation and are associated with object methods. An organization-wide policy specification, written in a high-level, domain-specific language based on XML [Bray et al. 98], specifies an access matrix [Lampson 71] that relates security identifiers to permissions, determining who can perform which operation. The policy also specifies the mapping between named resources and security identifiers, which determines the restrictions on named resources such as files, and the mapping between security operations and application code, which determines where to insert access checks into applications. The security service parses this policy and accordingly rewrites incoming applications, inserting calls to the enforcement manager at method and constructor boundaries so that resource accesses are preceded by the appropriate access checks. During execution of the rewritten application, the enforcement manager executes the inserted access checks, querying the security service based on the security identifiers and permissions it maintains. As a result, the security service performs the bulk of the functionality for policy processing and access mediation, reducing client-side checking to simple and efficient lookups. The client caches the results of security lookups for performance, and a cache-invalidation protocol between the security server and the enforcement manager enables the server to propagate changes in the access control matrix to clients. Policy changes that require new code paths to be instrumented, which we assume to be infrequent, require that applications be restarted using the remote administration service.

Our design addresses the two fundamental shortcomings of the security services in monolithic JVMs [Wallach et al. 97, Gong 97, Gong et al. 97]. First, in these systems the security policy is specified and enforced separately on each individual node. Consequently, a site administrator must maintain the security state for each node individually, which takes an effort proportional to the number of nodes. Second, the capability and stack-introspection schemes used in monolithic JVMs tie the security policy to the implementation of the system and therefore necessitate assistance from the original system developers to work. Essentially, the developers must anticipate the security requirements of the system and embed the requisite security checks (or hooks for such checks) at the appropriate places in the system libraries. In the case of Java, the developers of popular JVMs have anticipated and provided security checks on file, network and system property accesses. However, audio devices and window creation events have not been adequately protected, giving rise to denial-of-service attacks [McGraw & Felten 99]. In general, access control mechanisms that depend on a-priori anticipation and manual instrumentation are likely to be less flexible and less secure than systems that provide a clear separation between security policy and implementation, and allow the policy to be specified after deployment by users [Levin et al. 75, Erlingsson & Schneider 99].

3.3 Remote monitoring

System administration is particularly challenging for large networks of heterogeneous systems. Tasks such as resource accounting and usage pattern analysis, while easy to perform on time-shared systems of the past, are increasingly difficult to carry out in today's distributed systems. Our remote monitoring service allows administrators to track network-wide resource usage. Patterned after the security service, the remote monitoring service transforms applications to invoke auditing services at the appropriate places, such as on entry to and exit from method and constructor calls. As each application comes up, it contacts the remote monitoring console and a handshake protocol establishes the credentials of the user and assigns an identifier to the session. This connection is then used to send information to a central administration host that monitors client hardware configurations, users, JVM instances, code versions and noteworthy client events. Since the audit logs are stored on an external host that is not exposed to untrusted applications, a security breach may stop the creation of new audit events but cannot tamper with existing audit logs.

In addition to method-level remote monitoring services, we provide an instruction-level profiling and tracing service for monitoring application performance. The profiler instruments code to generate a dynamic call-graph [Graham et al. 82] from applications running across the network, as well as statistics on client code usage. We have used these services extensively both to debug our system and for optimizations. In particular, we have used the tracing service to obtain traces of synchronization behavior for Java applications and utilized this data in designing a transparent optimization service [Aldrich et al. 99].

3.4 Compilation

Compilation in monolithic virtual machines is typically accomplished by a just-in-time (JIT) compiler that translates bytecodes into native format as necessary. Since compilation is performed at the clients on demand, there are considerable time and resource pressures. Subsequently, JIT compilers typically do not perform aggressive optimizations [Proebsting et al. 97]. While newer compilation techniques explicitly address the time tradeoff between compilation, interpretation and execution by concentrating on the hot spots of a program [Sun 99], the fundamental difficulties posed by time constraints and lack of processing power on embedded hosts remain.

We are in the process of implementing a centralized network compiler that eliminates the burden of compilation from clients. As described in the previous section, the clients already perform a handshake with the remote administration service, in which they describe their native format. A compiler within the network can thus perform the translation for that platform ahead of time and thus amortize its startup costs over larger amounts of code. Resource investments in the compiler then benefit all clients in an organization.

3.5 Summary

The preceding discussion described the implementation of conventional virtual machine services in our architecture and illustrated the benefits of locating VM services at carefully chosen locations within the network. Centralization aids management and administration, provides a uniform interface to heterogeneous clients, and increases the integrity of the services by locating them on physically secure hosts. Partitioning conventional VM services into static and dynamic components makes it possible to perform much of the service functionality at a fixed cost when first loading an application.

4. Evaluation

Comparable performance to existing monolithic virtual machines is critical for a new system architecture to be adopted. This section shows that the performance of DVMs is as good as, and in some cases better than, monolithic virtual machines. More specifically, we evaluate the performance and scalability of our system. We show that end-to-end application performance within a distributed architecture is comparable to state of the art monolithic virtual machines. We also measure the throughput of distributed services under load and show that factored services do not present bottlenecks for networks of hundreds of clients.

Our performance comparisons were done using identical software and hardware platforms, but under different service architectures, to eliminate any biases introduced by client interpreters or compilers. In particular, we compare the running times of applications that have been transformed by our services to the run times of the original applications on Sun JDK 1.2, which is an industrial strength monolithic virtual machine. For our DVM performance measurements, we use the Sun JDK as a client but disable monolithic services such as security checking and rely on the injected code to provide the necessary service functionality. While we have also developed our own DVM client, which includes an interpreter, runtime, and garbage collector, all of our measurements are performed on the Sun JDK platform in order to provide a fair comparison.

All of the measurements in this section, including both client and server timings, were collected on 200 MHz PentiumPro systems running Windows NT4.0 SP3 with 64 MB of main memory, 256KB of cache, a Bus Logic SCSI driver, Seagate ST32151N disk drives, connected by a 10Mb/sec Ethernet through a 3C905 network interface. All numbers are the average of five runs, preceded by three throwaway runs to warm up any caches. The backbone connection to the Internet for our site is through two 100 Mb/sec links. At the time of the experiments, the peak consumed bandwidth on the links was less than 50 Mb/sec. We assume that the client-server network hop is physically secure and that security attacks are not initiated from within the organization, as is a common assumption with intranets or virtual private networks [Scott et al. 98]. Consequently, clients do not perform a digital signature check for each transferred class, but instead rely on the source host address for authentication.

4.1 Application performance

In this section, we compare distributed virtual machines to their monolithic counterparts and show that the runtime costs of distributed service components are comparable to monolithic services that offer the same functionality. Virtual machine services in our current implementation perform bytecode-to-bytecode transformations at the Java machine language level. Since the service-specific code snippets embedded into the applications must be type-safe and since the snippets are not integrated closely into the client JVM runtime, they may incur a higher performance overhead than their monolithic counterparts. In addition, the overhead of parsing, transforming and generating a modified binary at a network server can add extra latency to DVM operations. We first show that on end-to-end application benchmarks, the extra latency incurred by a DVM is small, even when servers have no more resources than clients. Second, we examine the overhead of the dynamic components for each service and show that DVMs can effectively partition work out of clients and reduce their computational requirements. Finally, we look at the overhead of the static service components and show that centralized services have only a modest impact on typical Internet fetch times.

Figure 5 shows the Java applications that we used for our end-to-end benchmarks. These applications were chosen because they are in common use, embody significant functionality, and range in complexity and size.

We first examine the end-to-end performance of these benchmarks under monolithic and distributed service architectures. The clients are arranged in a typical network setting, where a client pool is connected to a server, running Netscape Enterprise 2.01, through an HTTP proxy over

Name	Size	Classes	Description
JLex	91K	20	Lexical analyzer generator
Javacup	130K	35	LALR parser compiler
Pizza	825K	241	Bytecode to native compiler
Instantdb	312K	70	Relational database with a TPC-A like workload
Cassowary	85K	34	Constraint satisfier

Figure 5. Description of benchmark applications.

Ethernet. For DVMs, the proxy performs verification, security enforcement, and auditing. For monolithic virtual machines, the proxy acts as a null-proxy and the equivalent services are performed in the clients. For this experiment, we use a security policy and an audit specification that forces the DVM services to parse every class and examine every instruction in the applications. Figure 6 shows the running times of benchmark applications on monolithic VMs and DVMs. The first bar shows the execution time when all virtual machine services are performed by monolithic components embedded in the client. The second bar shows the execution time for an uncached execution of a benchmark under our distributed service infrastructure. Since DVMs require an extra step for parsing, instrumenting and generating a modified binary, applications will take longer to run on their first invocation. As expected, the overall execution time for DVMs is slightly slower, at 11% of total running time on average, than monolithic virtual machines. This difference is due largely to the overhead of parsing the application code in the proxy. The last bar shows the effects of proxy caching on the performance of distributed virtual machines. Following the first execution of an application by a host within the network, subsequent invocations run faster under

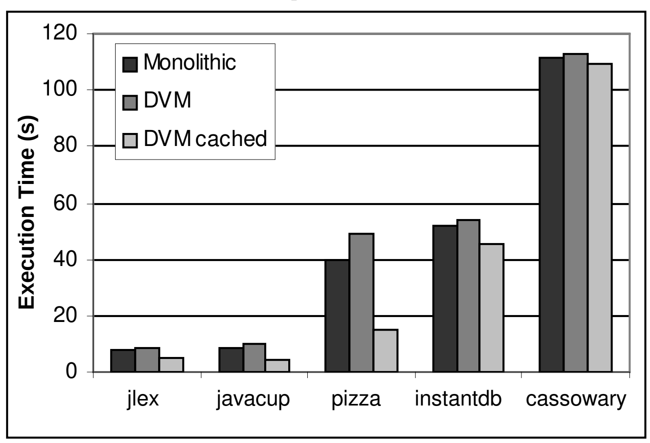

Figure 6. Application performance under monolithic and distributed virtual machines.

DVMs than monolithic virtual machines, because DVMs amortize the cost of performing services across all hosts in the organization and across multiple invocations by the same host.

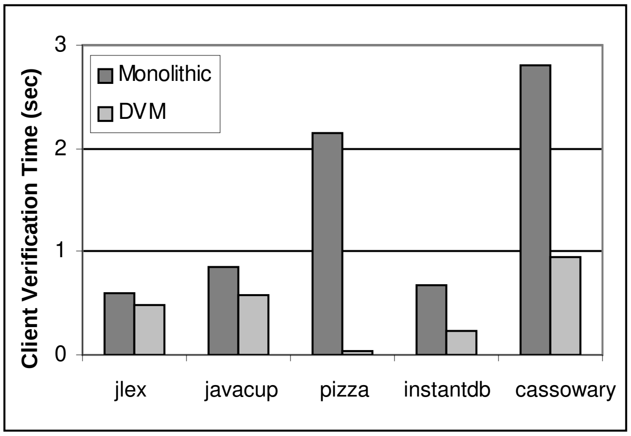

Figure 7. Client-side overhead for verification. Distributed virtual machines effectively decrease the load on the clients by factoring service functionality onto network servers.

4.1.1 Overhead of runtime checks

This section examines the amount of time clients spend executing system services and shows that the dynamic service components in DVMs are at least as fast as their monolithic counterparts. We first evaluate the dynamic component of the verification service. Figure 7 shows the time spent on verifying applications in monolithic and DVM clients by plotting the difference in total running time between unverified and verified applications. The DVM clients spend significantly less time on verification compared to monolithic VM clients. The self-verifying applications incur less client verification overhead because, as shown in Figure 8, the vast majority of verification checks have been performed statically. Hence, even though Sun's monolithic verifier is written in C and is tightly coupled with the rest of the virtual machine, self-verifying DVM applications run faster. In addition, since our verifier implementation resides on the network server, applying patches in response to security flaws takes a constant (and small) amount of effort under our architecture, whereas monolithic virtual machines often require time that is linear in the number of clients.

Benchmark	Static Checks	Dynamic Checks
Jlex	291679	371
Javacup	415825	806
Pizza	289495	541
Instantdb	1066944	3426
Cassowary	1965538	2346

Figure 8. Breakdown of static and dynamic checks performed by the verifier. The vast majority of checks occur at the network server, prior to execution, thereby improving client performance.

Next, we examine the security service and show that DVMs enable network-wide enforcement of security constraints without slowing down the common case of security checking. We use a set of security microbenchmarks to compare the stack introspection-based access control scheme [Gong & Schemers 98] implemented in Sun JDK1.2 to our own security service described earlier. Figure 9 summarizes the performance of our security microbenchmarks under the two different service architectures. The benchmarks perform the system resource accesses described in the first column under a policy that permits the access. The baseline shows the latency of the unchecked (insecure) Java operation, which ranges from a few microseconds to a few milliseconds. The columns labeled "check" show the expired wall clock time for checked operations, where the credentials of the thread performing the operation are checked against the security policy. The columns labeled "overhead" present the difference between the baseline and the checked operation, and show the amount of time spent in the security service. The first security check in our service, shown separately in the "download" column, incurs the cost of downloading a portion of the global security policy from a server. Although this has high overhead, it occurs infrequently and accounts for a small fraction of the total running time of most applications. The performance of subsequent checks, however, is generally comparable to the Sun JDK. While our security service is slow when checking thread operations due to an object identifier lookup, for GetProperty and OpenFile benchmarks, our security service

Description	Baseline (no check)	JDK (check)	JDK (overhead)	DVM (download)	DVM (check)	DVM (overhead)
Get Property	0.0020	0.0488	0.0468	5.830	0.0092	0.0072
Open File	1.406	8.631	7.224	6.406	1.430	0.0238
Change Thread Priority	0.0638	0.0645	0.0007	5.026	0.0815	0.0177
Read File	0.0141	N/A	N/A	4.146	0.0368	0.0227

Figure 9. Performance of security services on monolithic and distributed virtual machines. Times are in milliseconds. Distributed virtual machines enable security enforcement at locations, such as file read, not foreseen by system designers. The common case of security checking has comparable overhead to monolithic virtual machines.

outperforms the Sun JDK implementation by a factor of 7 and 300, respectively.

Qualitatively, our security service is more versatile than the monolithic service in the Sun JDK because it can impose security checks at any point in the system libraries, whereas the Sun JDK implementation can only impose checks at predetermined locations in the code. Consequently, it cannot adequately protect file read operations, as these operations do not contain security checks. A malicious application that acquires a file handle to an open file through a leaky interface can thus avoid security checks, which are imposed only on object creation.

4.1.2 Overhead of the proxy

Finally, we examine the overhead of binary rewriting on class transfer latency and show that the overhead of the static service components is small compared to typical wide-area transfer latencies. To measure the overhead associated with downloading Java applets through our service infrastructure, we collected a list of all indexed Java applets from the AltaVista search engine and randomly selected a subset of 100 applets. The average latency of downloading an applet from the Internet is 2198 ms, with a large standard deviation of 3752 ms. When the applet is not cached, our proxy adds about 265 ms of processing time to parse and instrument it, which amounts to a 12% overhead over the average load latency. This overhead can be masked, if necessary, by using a non-strict execution model in the clients [Krintz et al. 98]. Accesses to classes that have been fetched by another DVM client are served from an on-disk cache on the proxy and take only 338 ms to download on average.

4.2 Scaling

While DVMs offer increased manageability and integrity by factoring services out of clients, the centralized network servers could potentially form bottlenecks. We examine the scaling characteristics of the static service components for large numbers of clients (Figure 10) under the worst possible scenario and show that a central proxy server does not form a bottleneck for networks of hundreds of active virtual machines. In this experiment, up to 250 clients *simultaneously* fetch different applets from the Internet through our proxy, with proxy caching disabled. The graph shows the total sustained proxy throughput as a function of the number of clients. Since the static service components can defer parts of service functionality to run time, they do not inherently need to synchronize with clients or require exclusive access to shared state. Consequently, the throughput scales linearly for up to 250 simultaneous clients and degrades thereafter as all 64MB of server memory are used up. The average client fetch latency per kilobyte is roughly constant, between 1.0 and 1.2 sec/kB, throughout the range of 50-250 clients where there are statistically significant samples. Based on these numbers,

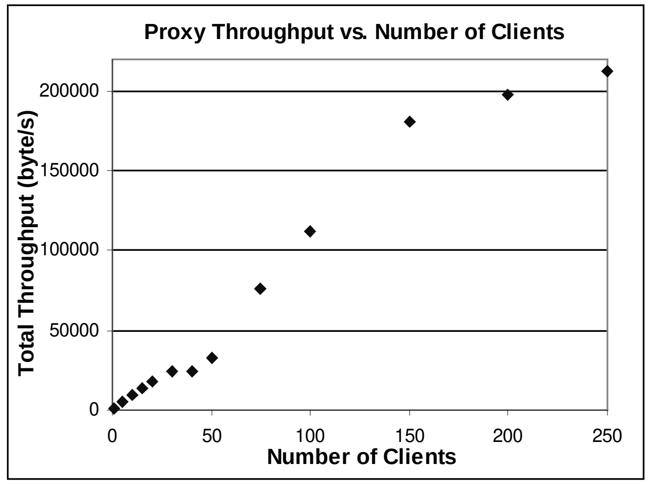

Figure 10. Sustained throughput of our VM services versus number of clients.

we conclude that the proxy does not pose a bottleneck for networks of hundreds of clients. Note that this experiment represents a worst-case scenario for a DVM, as caching was disabled in the proxy in order to not predicate the good performance of the system on the cacheability of classes and locality of accesses. In larger installations, an administrator can rely on the fact that average loads will be a fraction of the worst case load, enable caching, add more memory to the proxy, or use replicated proxies.

4.3 Summary

Our DVM system achieves end-to-end performance that is comparable to state of the art monolithic virtual machines. Service partitioning between static and dynamic components factors substantial work out of clients and avoids expensive network communication. Further, service-specific code injection enables the run-time functionality to be tailored to application needs. For example, an earlier implementation of our verifier relied on reflection primitives built into the JVM and was too slow. We subsequently developed a reflection service that adds self-describing attributes to classes and modified our verifier to use this interface rather than the slow library interface in the Sun JDK. While anecdotal, this example demonstrates that binary rewriting services can effectively be used to compensate for limitations in client performance and functionality. Overall, binary rewriting is a general and flexible technique for implementing services in a distributed virtual machine.

5. Optimizations for mobile code and low bandwidth links

The preceding section discussed existing virtual machine services and compared them to their monolithic counterparts. In this section, we show how our architecture provides a platform for performing optimizations, such as client-specific code modifications, that are not supported by monolithic virtual machines. Specifically, we examine a scenario where some clients, such as handheld computers or wireless PDAs, are connected to a common network via low bandwidth wireless links. While these clients may present the same external VM interfaces as a desktop computer, they need to utilize the available network bandwidth effectively in order to decrease time spent loading applications from the network.

State of the art Java virtual machines, even though originally targeted at embedded systems, pay little attention to optimizing network transfers or to developing binary distribution formats that maximize effective network bandwidth. Java offers two separate modes of transport, one in which the whole application is shipped as a single unit, and another where entire object implementations are fetched at first reference to any method or field of the object. Even in the latter case, where lazy object loading filters out unused object definitions, roughly 10-30% of all downloaded code is never invoked [Sirer et al. 99].

Fundamentally, the problem is that the units of code distribution in Java are not suitable for efficient bandwidth utilization. The granularity at which code is transferred corresponds either to source-derived logical abstractions, such as classes, or artifacts of the compilation and linking process, such as Java archive (JAR) files. These coarse units of code transfer fail to capture the dynamic execution path for an application. A client must request the complete class implementation even when it requires only a single method from the class during the entire execution. As a result, the clients incur runtime costs, which include delays in program startup and execution from transfer of unused code, increased memory consumption from storing unused components and interference with other threads of execution which share or serialize on common resources during the transfer.

To help solve these problems, we have developed a practical service based on DVMs for restructuring mobile Java applications. We leverage the pipeline described in section 2 to introduce a separate optimization step that takes place on a network server. Here, application code is split up into smaller transfer units, based on a profile, to use the available network bandwidth more effectively for downloading programs. This repartitioning service uses binary rewriting to restructure application components at method granularity such that frequently used and related methods are grouped together, while less frequently used methods are factored out into separate units. This repartitioning is performed on the fly within the network. The network proxy collects profile information from the first execution of an application and uses the profile to generate a first-use graph of the methods in the application. This graph is then used to partition unused methods into separate classes that are loaded only on demand. Neither the JVM clients nor the web servers that provide the application code need to be modified to support this service. The DVM architecture provides a convenient location in the network for performing transparent code modifications and enables client-specific optimizations without leaking information about client properties outside their administrative domain.

Figure 11 plots the startup times of a set of graphical Java applets and applications as a function of bandwidth and shows that effective use of bandwidth is crucial for application performance when the clients are connected over slow (less than 1Mb/sec) links. We define startup time as the time from initial invocation to the time when the application can start processing user requests and examine it because it contributes to unmaskable user delays. Figure 12 shows the performance impact of our optimization service. The optimization service achieves speedups of up to 28% in application performance over 28.8 Kb/sec links via code repartitioning within the network.

The performance gains possible with code repartitioning show the utility of service platforms located within the network. While this particular service reduces code transfer time and decreases memory requirements on

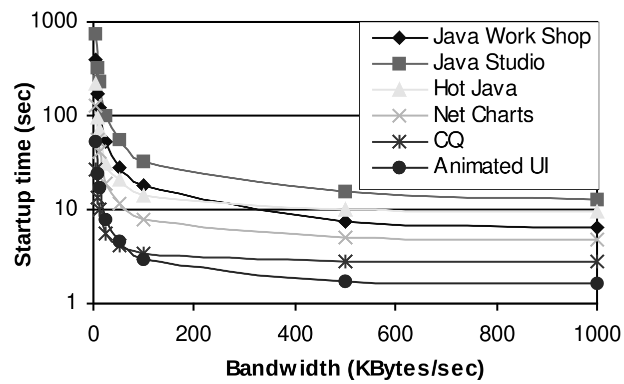

Figure 11. Application start-up time as a function of network bandwidth.

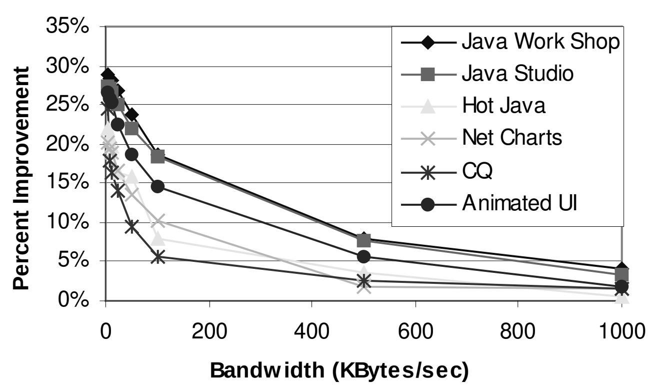

Figure 12. Percent improvement in application start-up time with client-specific optimization services.

clients, it also exemplifies a class of client-specific optimizations that can be performed transparently within the network under a distributed virtual machine architecture.

6. Related work

While virtual machines have evolved considerably since their introduction, their monolithic architecture has remained unchanged. An important early virtual machine was the IBM VM system [IBMVM 86], which made its commercial debut in 1972. The IBM VM system enabled organizations to run both the MVS and CMS operating systems on the same physical machine by virtualizing machine resources. The monolithic service architecture of this system meant that all VM services were executed on the same host [Deitel 89]. This structure has influenced many of the virtual machine implementations that followed.

In the early 1970's, virtual machines were adopted by the language community as a substrate for code distribution. These systems used virtual machines both to retain the portability of applications in an increasingly heterogeneous environment and to provide high-level abstractions for which compilers could generate code more easily. P-Code [UCI 96] is one such intermediate language that was widely used as part of the UCSD Pascal system. It offered cross-platform portability by targeting a pseudo-machine for compilation and relying on a runtime interpreter. The P-System concentrated all virtual machine functionality on its clients.

Recent virtual machines such as Java and Inferno build on this legacy to provide safety, portability and uniformity in a network computing setting. They rely on a growing set of complex services [Gosling & Yellin 96, Myers & Liskov 97, Wallach et al. 97], located and executed on the clients. This trend exacerbates the problems associated with monolithic virtual machines outlined in this paper.

The playground approach described in [Malkhi et al. 98] as well as the CAGE system from Digitivity take an alternative approach to factoring services out of clients. These systems replace all virtual machines in an organization with a single virtual machine that is physically isolated behind a firewall. All applications execute on this centralized virtual machine and communicate with clients only for user-interface operations. While this approach has the property that clients are not exposed to untrusted code, it prevents the secure sharing of client resources, including files and peripherals. Consequently, while it may be applicable to stateless applets on the web, it does not support applications that require selective access to local resources found on clients.

Service centralization has also been effective in the World Wide Web. Notably, Fox et al. define an architecture for transformation, aggregation, caching and customization (TACC) of non-executable Internet content [Fox et al. 97]. TACC services are centralized in a cluster-based proxy and web pages are specialized according to the needs of clients. DVMs also perform transformation, caching and customization services, but operate on executable code instead of static content. Furthermore, static service transformations in a DVM are driven not only by the needs of clients but also by the needs of the overall organization.

Several projects have explored the use of binary rewriting for Java. JOIE [Cohen et al. 98] provides a general framework for modifying Java classes during load-time. Similarly, BIT [Lee & Zorn 97] is an event-based binary instrumentation tool that operates on Java class files. These tools are comparable to the binary rewriting engine on which our services are based. JRes [Czajkowski & von Eicken 98] uses binary rewriting to provide resource accounting for memory, CPU time and network resources in Java. Naccio [Evans & Twyman 99] uses binary rewriting to impose security and resource usage policies on applications. These systems work at the level of individual JVMs and lack a centralized infrastructure as discussed in this paper. The rewriting services they offer are complementary to, and would be accommodated by, the distributed service architecture described here.

An alternative for enforcing security constraints on networks of JVMs is to give up on the Java security interfaces altogether and to instead rely on the security features, such as file-system permissions, provided by the underlying operating system. This approach is undesirable for three separate reasons. First, there may not be an operating system beneath the virtual machine, as there is no requirement that a VM be layered on a more powerful substrate. Second, there may be many different types and versions of operating systems underneath virtual machines, making the administration of a uniform security policy difficult in a heterogeneous network [Hitz et al. 98]. Finally, access controls provided by general-purpose operating systems can not distinguish between multiple applications executing within the same VM, because these abstractions are only visible within the JVM. Overall, security enforcement needs to be performed at the JVM level in order to be universally applicable, present a uniform interface and provide fine-grain security controls.

7. Conclusions

We have designed and implemented a new system architecture for network computing based on distributed virtual machines. Our system factors virtual machine services out of clients and locates them in organization-wide network servers. The services operate by intercepting application code and modifying it on the fly to provide service functionality. This paper shows that distributed virtual machines can reduce client resource requirements, simplify management and isolate security-critical services from untrusted and potentially malicious code. Our particular service implementation strategy is based on factoring VM services out of clients, partitioning them into static and dynamic components, and implementing them through binary rewriting. This approach supports diverse VM services with comparable performance to monolithic virtual machines.

Acknowledgements

We would like to thank Nathan R. Anderson, Sean McDirmid and Bibek Pandey for their assistance with parts of our DVM implementation and performance evaluation. We would also like to thank our shepherd, Fred Schneider, for his guidance and detailed suggestions, as well as the anonymous reviewers for their helpful comments.

References

[Accetta et al. 86] M. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young. Mach: A New Kernel Foundation for Unix Development. In *Proceedings of the USENIX Summer Conference*, pages 93–112, Atlanta, Georgia, June 1986.

[Aldrich et al. 99] J. Aldrich, C. Chambers, E. G. Sirer and S. Eggers. Optimizing Unnecessary Synchronization Operations from Java Programs. In *Static Analyses Symposium*, pages 19–38, Venice, Italy, September 1999.

[Bershad et al. 95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 267–284, Copper Mountain Resort, Colorado, December 1995.

[Bray et al. 98] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, editors. *Extensible Markup Language (XML) 1.0*. W3C Recommendation, World Wide Web Consortium, February 1998.

[CERT 96] CERT Coordination Center. Weaknesses in Java Bytecode Verifier. September 1999. http://www.cert.org/advisories/CA-96.07.java_bytecode_verifier.html.

[Cheswick & Bellowin 94] W. R. Cheswick and S. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, June 1994.

[Cohen 97] R. M. Cohen. *The Defensive Java Virtual Machine Specification*. Computational Logic Inc., May 1997. http://www.cli.com/software/djvm/.

[Cohen et al. 98] G. A. Cohen, J. S. Chase, and D. L. Kaminsky. Automatic Program Transformation with JOIE. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 167–178, New Orleans, Louisiana, 1998.

[Czajkowski & von Eicken 98] G. Czajkowski and T. von Eicken. JRes: A Resource Accounting Interface for Java. In *Proceedings of the OOPSLA'98 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 21–35, Vancouver, Canada, October 1998.

[Dean et al. 97] D. Dean, E. W. Felten, D. S. Wallach, and D. Belfanz. Java Security: Web Browers and Beyond. In D. E. Denning and P. J. Denning, editors, *Internet Beseiged: Countering Cyberspace Scofflaws*, pages 241–269. ACM Press, October 1997.

[Deitel 89] H. M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley, August 1989.

[Dorward et al. 97] S. Dorward, R. Pike, D. L. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. The Inferno Operating System. *Bell Labs Technical Journal*, 2(1):5–18, Winter 1997.

[Engler et al. 95] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[Erlingsson & Schneider 99] U. Erlingsson and F. B. Schneider. SASI Enforcement of Security Policies: A Retrospective. Cornell Technical Report TR99-1758. Ithaca, New York, 1999.

[Evans & Twyman 99] D. Evans and A. Twyman. Flexible Policy-Directed Code Safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 32–45, Oakland, California, May 1999.

[Fox et al. 97] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 78–91, Saint-Malo, France, October 1997.

[Freund & Mitchell 98] S. N. Freund and J. C. Mitchell. A Type System for Object Initialization in the Java Bytecode Language. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 310–327, Vancouver, Canada, October 1998.

[Goldberg 73] R. P. Goldberg. *Architectural Principles for Virtual Computer Systems*. Ph.D. thesis, Harvard University, 1973.

[Gong 97] L. Gong. Java Security: Present and Near Future. *IEEE Micro*, 17(3):14–19,May/June 1997.

[Gong 99] L. Gong. *Inside Java 2 Platform Security—Architecture, API Design, and Implementation*. Addison-Wesley, 1999.

[Gong et al. 97] L. Gong, M. Mueller, L. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An

Overview of the New Security Architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, California, December 1997.

[Gong & Schemers 98] L. Gong and R. Schemers. Implementing Protection Domains in the Java Development Kit 1.2. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, pages 125–134, San Diego, California, March 1998.

[Gosling & Yellin 96] J. Gosling and F. Yellin. *The Java Application Programming Interface*, Volumes 1 & 2. Addison-Wesley, 1996.

[Graham et al. 82] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A Call Graph Execution Profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, *SIGPLAN Notices*, 17(6):120–126, June 1982.

[Grimm & Bershad 99] R. Grimm and B. N. Bershad. Providing Policy-Neutral and Transparent Access Control in Extensible Systems. In J. Vitek and C. D. Jensen, editors, *Secure Internet Programming—Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 317–338, Springer, 1999.

[Heidemann & Popek 94] J. S. Heidemann and G. J. Popek. File-System Development with Stackable Layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[Hennessy 99] J. Hennessy. The Future of Systems Research. In *IEEE Computer*, pages 27–33, August 1999.

[HP 99] Hewlett-Packard Company. *Chai Products*. http://www.hp.com/emso/products/.

[Hitz et al. 98] D. Hitz, B. Allison, A. Borr, R. Hawley, and M. Muhlestein. Merging NT and Unix Filesystem Permissions. In *Proceedings of the Second Usenix Windows NT Symposium*, pages 87–95, Seattle, Washington, August 1998.

[IBMVM 86] IBM Corporation. Virtual Machine/System Product Application Development Guide, Release 5. Endicott, New York, 1986.

[ISC 99] Internet Software Consortium. Internet Host Domain Survey. July 1999, http://www.isc.org/.

[Krintz et al. 98] C. Krintz, B. Calder, H. P. Lee and B. G. Zorn. Overlapping Execution with Transfer Using Non-Strict Execution for Mobile Programs. In the *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 159–169, San Jose, California, October 1998.

[Lampson 71] B. Lampson. Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, pages 437-443, Princeton, Ney Jersey, March 1971. Reprinted *in Operating Systems Review*, 8(1):18–24, January 1974.

[Lee & Zorn 97] H. Lee and B. Zorn. BIT: A Tool for Instrumenting Java Bytecodes. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 73–82, Monterey, California, December 1997.

[Levin et al. 75] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/Mechanism Separation in Hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, pages 132–140, Austin, Texas, October 1975.

[Lindholm & Yellin 96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, September 1996.

[Malkhi et al. 98] D. Malkhi, M. K. Reiter, and A. D. Rubin. Secure Execution of Java Applets using a Remote Playground. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 40–51, Oakland, California, May 1998.

[McGraw & Felten 96] G. McGraw and E. W. Felten. *Java Security: Hostile Applets, Holes and Antidotes*. Wiley Computer Publishing, John Wiley & Sons, December 1996.

[McGraw & Felten 99] G. McGraw and E. W. Felten. *Securing Java: Getting Down to Business With Mobile Code*. Wiley Computer Publishing, John Wiley & Sons, February 1999.

[Minear 95] S. E. Minear. Providing Policy Control Over Object Operations in a Mach Based System. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 141–156, Salt Lake City, Utah, June 1995.

[Mogul 89] J. C. Mogul. Simple and Flexible Datagram Access Controls for Unix-based Gateways. In *Proceedings of the Summer 1989 USENIX Conference*, pages 203–221, Baltimore, Maryland, June 1989.

[Myers & Liskov 97] A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 129–142, Saint-Malo, France, October 1997.

[Olawsky et al. 96] D. Olawsky, T. Fine, E. Schneider, and R. Spencer. Developing and Using a "Policy Neutral" Access Control Policy. In *Proceedings of the New Security Paradigms Workshop*, September 1996.

[O'Malley & Peterson 92] S. W. O'Malley and L. L. Peterson. A Dynamic Network Architecture. *ACM*

Transactions on Computer Systems, 10(2):110–143, May 1992.

[Popek & Goldberg 74] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. In *Communications of the ACM*, 17(7):412–421, July 1974.

[Proebsting et al. 97] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson. Toba: Java for Applications, A Way Ahead of Time (WAT) Compiler. In *Proceedings of the Third Conference on Object-Oriented Technologies and Systems*, pages 41–53, Berkeley, California, June 1997.

[Rayside et al. 98] D. Rayside, S. Kerr and K. Kontogiannis. Change And Adaptive Maintenance in Java Software Systems. In *Fifth Working Conference on Reverse Engineering*, pages 10–19, Honolulu, Hawaii, October 1998.

[Rivest et al. 78] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[Rivest 92] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, Internet Engineering Task Force, April 1992.

[Saltzer & Shroeder 75] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[SCC 97] Secure Computing Corporation. DTOS Lessons Learned Report. Technical Report DTOS CDRL A008, Secure Computing Corporation, Roseville, Minnesota, June 1997.

[Scott et al. 98] C. Scott, P. Wolfe and M. Erwin. *Virtual Private Networks*. O'Reilly, December 1998.

[Seltzer 98] L. Seltzer. Java Environments. *PC Magazine*, pages 137–141, April 7, 1998.

[Sirer et al. 99] E. G. Sirer, A. J. Gregory and B. N. Bershad. A Practical Approach for Improving Startup Latency in Java Applications. In *Workshop on Compiler Support for Systems Software*, INRIA Technical Report #0228, pages 47–55, Grenoble, France, May 1999.

[Sirer & Bershad 99] E. G. Sirer and B. N. Bershad. Using Production Grammars in Software Testing. In *Second Conference on Domain Specific Languages*, pages 1–13, Austin, Texas, October 1999.

[Spyglass 98] Spyglass, Inc.Software Development Interface. Naperville, IL, September 1999. http://www.spyglass.com/products/smosaic/sdi/sdi_spec.html

[Stata & Abadi 98] R. Stata and M. Abadi. A Type System for Java Bytecode Subroutines. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 149–160, San Diego, California, January 1998.

[Sun 99] Sun Microsystems. The Java HotSpot Performance Engine Architecture. White paper, Sun Microsystems, Palo Alto, California, September 1999. http://java.sun.com/products/hotspot/whitepaper.html.

[Thorn 97] T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, pages 213–239, 29(3), September 1997.

[UCI 96] University of California, Irvine. p-System: Description, Background, Utilities. http://www.ics.uci.edu/~archive/documentation/p-system/p-system.html.

[Wallach et al. 97] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible Security Architectures for Java. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 116–128, Saint-Malo, France, October 1997.

[Webb 99] W. Webb. Embedded Java: An Uncertain Future. *Electrical Design News*, 44(10):89–96, May 1999.

[Weiser 93] M. Weiser. Some Computer Science Problems in Ubiquitous Computing. In *Communications of the ACM*, pages 74–84, July 1993.