Libra: A Library Operating System for a JVM in a Virtualized Execution Environment

Glenn Ammons
IBM T.J. Watson Research Center
ammons@us.ibm.com

Jonathan Appavoo
IBM T.J. Watson Research Center
jappavoo@us.ibm.com

Maria Butrico
IBM T.J. Watson Research Center
butrico@us.ibm.com

Dilma Da Silva
IBM T.J. Watson Research Center
dilmasilva@us.ibm.com

David Grove
IBM T.J. Watson Research Center
groved@us.ibm.com

Kiyokuni Kawachiya
IBM Tokyo Research Laboratory
kawatiya@jp.ibm.com

Orran Krieger
IBM T.J. Watson Research Center
okrieg@us.ibm.com

Byran Rosenburg
IBM T.J. Watson Research Center
rosnbrg@us.ibm.com

Eric Van Hensbergen
IBM Austin Research Laboratory
ericvanhensbergen@us.ibm.com

Robert W. Wisniewski
IBM T.J. Watson Research Center
bobww@us.ibm.com

Abstract

If the operating system could be specialized for every application, many applications would run faster. For example, Java virtual machines (JVMs) provide their own threading model and memory protection, so general-purpose operating system implementations of these abstractions are redundant. However, traditional means of transforming existing systems into specialized systems are difficult to adopt because they require replacing the entire operating system.

This paper describes Libra, an execution environment specialized for IBM's J9 JVM. Libra does not replace the entire operating system. Instead, Libra and J9 form a single statically-linked image that runs in a hypervisor partition. Libra provides the services necessary to achieve good performance for the Java workloads of interest but relies on an instance of Linux in another hypervisor partition to provide a networking stack, a filesystem, and other services. The expense of remote calls is offset by the fact that Libra's services can be customized for a particular workload; for example, on the Nutch search engine, we show that two simple customizations improve application throughput by a factor of 2.7.

Categories and Subject Descriptors D.3.4 [*Processors*]: Runtime Environments; D.4.7 [*Operating Systems*]: Organization and Design; D.4.8 [*Operating Systems*]: Performance

General Terms Design, Experimentation, Performance

Keywords Virtualization, exokernels, Xen, JVM

VEE '07 June 13–15, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-630-1/07/0006...$5.00

1. Introduction

This paper describes a new way to transform existing software systems into high-performance, specialized systems. Our method relies on hypervisors [12, 14], which are becoming efficient and widely available, and on the 9P distributed filesystem protocol [30, 32].

Our approach is similar to the exokernel approach [25]. An exokernel system divides the general-purpose operating system into two parts: a small, trusted kernel (called the exokernel) that securely multiplexes hardware resources such as processors and disk blocks, and a collection of unprivileged libraries (called "library operating systems" or "libOSes") that provide operating system abstractions such as filesystems and processes. Ideally, each application tailors the abstractions to its needs and pays only for what it uses. For example, the distributed search application Nutch [8, 9] needs a Java virtual machine, access to a read-only store, and a simple networking stack; Section 5.3 shows that simple implementations of these abstractions achieve good performance.

Unfortunately, exokernels are difficult to adopt, because migrating an existing application to an exokernel system requires porting the operating system on which it relies. For example, to run unmodified UNIX programs on their exokernel, Kaashoek and others wrote ExOS, a library that implements many of the BSD 4.4 abstractions [25]. Writing such a library is a significant effort. Also, because the library is a reimplementation of the operating system, the only way to take advantage of improvements to the operating system is to port them to the library.

Our system, Libra,[1] avoids these problems by casting a hypervisor (specifically, Xen [5]) in the role of the exokernel. Figure 1 depicts the overall architecture of Libra. Unlike traditional exoker-

[1] We chose the name "Libra" because our goal of providing well-balanced services aligns with the imagery associated with the constellation of the same name.

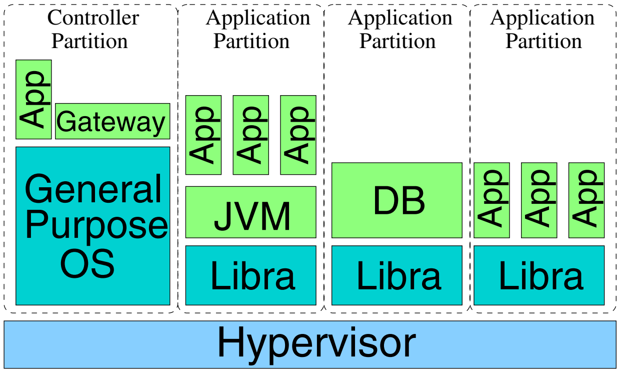

Figure 1. Proposed system architecture.

nels, hypervisors run other operating systems with few or no modifications [27, 5, 42]. By running an operating system (the controller) in a hypervisor partition, applications can be migrated incrementally to an exokernel system. In the first step of the migration, the application runs in its own partition instead of as an operating system process but accesses system services on the controller remotely through the 9P distributed filesystem protocol [32]. Later, as needed, these relatively expensive remote calls are replaced by efficient library implementations of the same abstractions.

Proponents of exokernels object to hypervisors because "[hypervisors] confine specialized operating systems and associated processes to isolated virtual machines...sacrificing a single view of the machine" [25]. However, in our approach, a controller partition provides a single view of the machine: applications running in other partitions correspond to processes at the controller. This is true even for applications that run on other machines in a cluster. Section 2 explains how Libra achieves this unified machine view.

This paper makes three main contributions:

- The hypervisor approach for transforming existing systems into high-performance, specialized systems.
- A case study in which Nutch, a large distributed application that includes a commercial JVM (IBM's J9 [4]), is transformed into such a system.
- Examples of Nutch-specific optimizations that result in good performance.

The rest of the paper is organized as follows. Section 2 explains the overall architecture we are proposing. Section 3 describes the implementation of Libra and our port of J9 to Libra. Details of our port of Nutch to J9/Libra, including specializations that improve its throughput, are described in Section 4. Section 5 evaluates J9/Libra's performance on various benchmarks, including Nutch, the standard SPECjvm98 [39] and SPECjbb2000 [38] benchmarks, and two microbenchmarks. Section 6 discusses related work. Finally, Section 7 outlines future work and Section 8 concludes the paper.

2. Design

This section explains our proposed architecture, which is depicted in Figure 1. At the bottom of the software stack, a hypervisor hosts a controller partition and one or more application partitions. The hypervisor interacts directly with the hardware to provision hardware resources among partitions, providing high-level memory, processor, and device management.

Above the hypervisor, a general-purpose operating system such as Linux runs as a "controller" partition. The controller is the point of administrative control for the system and provides a familiar environment for both users and applications. Each application partition is launched from the controller by a script that invokes the hypervisor to create a new partition and load the application into it. This script also launches a gateway server that permits the application to access the controller's resources, services, and environment.

The gateway server is an extended version of Inferno [31], which is a compact operating system that can run on other operating systems. Inferno creates a private, file-like namespace that contains services such as the user's console, the controller's filesystem, and the network (see Figure 2). The application accesses this namespace remotely via the 9P resource sharing protocol [30], which runs over a shared-memory transport established between the controller and application partitions. A more detailed description of our extensions to Inferno and a preliminary performance analysis of the transport is available in another paper [20].

Note that nothing in the architecture requires applications to access *all* resources through the gateway, because the hypervisor allows resources and peripherals to be dedicated to an application partition and accessed directly. Alternatively, applications can use 9P to access resources across the network, either directly or through the gateway. Facilities for redundant resource servers, fail over, and automated recovery have also been explored [16].

As in an exokernel system, each application is linked with a library operating system (libOS). However, unlike in exokernel systems, the libOS focuses only on performance-critical services; other services are obtained from the controller through the gateway. For example, Libra, the libOS described in this paper, contains a thread library implementation but accesses files remotely.

This approach not only reduces the cost of developing a libOS but also reduces the cost of administering new partitions. Because applications share filesystems, network configuration, and system configuration with the controller, administering an additional application partition is cheaper than administering an additional operating system partition. Also, because 9P gateways run as the user who launched the application, they inherit the same permissions and limitations, taking advantage of existing security mechanisms and resource policies.

Finally, because we use a hypervisor instead of an exokernel to multiplex system resources, applications can use privileged execution modes and instructions. This enables optimizations and eases migration, because a libOS can be simply a pared-down, general-purpose operating system. The combination of supervisor-mode capability and 9P for access to remote services is flexible: application partitions can be like traditional operating systems, self-contained with statically-allocated hardware resources; like microkernel applications, with system services spread across various protection domains; or like a hybrid of the two architectures that makes sense for the particular workload being executed.

3. J9/Libra Implementation

This section describes the implementation of Libra and the port of the J9 [4] virtual machine to this new platform. It was a somewhat atypical porting process, since we were simultaneously porting J9 to the Libra abstractions while designing and implementing new Libra abstractions to support J9's execution. We begin by introducing the relevant aspects of J9 and briefly describing our porting and debugging methodology. Next, we discuss the major Libra subsystems needed by J9. Finally we highlight the major limitations of our current implementation.

3.1 J9 Overview

J9 is one of IBM's production JVMs and is deployed on more than a dozen major platforms ranging from cell phones to zSeries mainframes. Because it runs on such a diverse set of platforms, a great deal of effort has been invested by the J9 team in defining

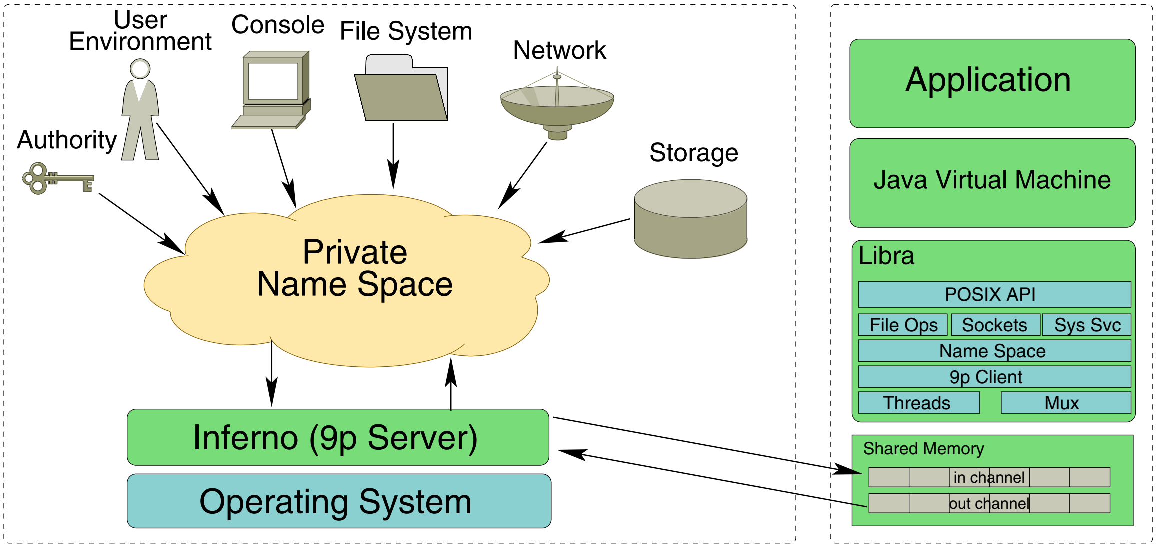

Figure 2. Resource organization and sharing.

a portability layer, called the *port library*, and ensuring that the JVM, JIT, and core class library native code only access platform-dependent APIs through this abstraction layer.

The port library proper consists of approximately 175 functions that provide APIs to the filesystem, network, memory management, and miscellaneous system services such as timing and hardware cache management. In addition, J9 depends on a thread library that defines approximately 100 thread and synchronization related functions that serve to encapsulate platform-specific APIs. In addition to these two formal porting abstractions, the J9 code base also makes direct use of some fundamental `libc` functions such as `memcpy`, `isdigit`, and `qsort` that are generally available with consistent semantics across a wide range of platforms.

The fact that J9 had already been ported to such a diverse set of environments significantly simplified our task of porting it to Libra. Although we eventually ended up running configurations of J9 that included well over half a million lines of C, C++, and assembly code, we had to modify less than 20 files outside of the portability layer.

3.2 Porting Methodology

The port of J9 to Libra was an iterative process: we began with the simplest and smallest possible configuration of J9 we could define for Linux/PowerPC64 and got it to run "HelloWorld" on the simplest possible incarnation of Libra. In this initial stage we were using a custom configuration of J9 that used the `CLDC` J2ME class libraries [40], had no JIT compiler, and disabled all optional VM features such as RAS and trace tooling. We then iteratively enabled the disabled J9 functionality and worked on running larger workloads with more extensive class libraries. As the workloads got more complex, we extended Libra and its interactions with the controller partition to support the necessary functionality. We are currently running with full JIT compilation and using the largest set of IBM authored class libraries available for our version of J9.[2]

We began the porting process by defining dummy implementations of all the functions in the port and thread libraries. The implementation of these functions simply printed the name of the function and invoked the debugger. We provided a similar stub implementation of all `libc` functions that were statically referenced by J9. Discovering these functions was straightforward, as our build process builds all of J9 into a single `.o` that is statically linked against Libra without the standard C library to produce an executable. Trivial functions were implemented immediately, but in most cases we implemented them on-demand as the tripping of an assertion indicated that the function was dynamically needed.

Although we are now able to execute Java workloads, including SPECjbb2000, SPECjvm98, and Nutch, we still have not fully implemented all of the stubbed out port library functions. Of the functions that remain unimplemented, 50% support socket and network operations. Most of the rest support JVM functionality that we have not yet re-enabled, such as signal handling and shared memory regions.

3.3 Debugging J9 on Libra

Throughout this process we attempted to maintain J9/Libra and J9/Linux configurations that were as similar as possible. These dual configurations enabled a debugging approach in which we could run a program that failed or crashed on J9/Libra on J9/Linux and see where the failing and successful executions diverged. In the early stages of the process, we relied heavily on logging J9's interactions with the port and thread libraries to discover where executions diverged. As the system became more robust, and the bugs became subtler, we relied more on gdb-remote debugging to investigate J9 crashes. Libra implements gdb stub functions whose inputs and outputs are redirected by the gateway server to a TCP port. By using this mechanism, J9 and Libra can be debugged using gdb's remote debugging functions [37].

3.4 Libra Subsystems

The services that J9 requires from the underlying system fall into four main categories: memory management, filesystem access, thread support, and socket support. A few miscellaneous support routines that fall outside these categories (e.g. for access to environment variables and to the system clock) are also needed.

Libra is given a memory partition when it starts, and currently that partition neither grows nor shrinks. Part of the partition is occupied by the program text and data of the J9/Libra image. The rest is free and must be managed to satisfy the dynamic memory requirements of J9 and of Libra itself. We're currently using a simple two-level management hierarchy. Page-granularity free memory is

[2] These are a subset of the Java 1.4 J2SE libraries.

tracked with a bit vector; smaller chunks are rounded to power-of-two sizes and are managed via free-lists. J9 requests memory in large chunks for its heap. In a general-purpose operating system environment, these requests reserve ranges of virtual address space, with the actual memory allocation happening later as the space is first used. In Libra, memory resources are allocated at the time they are requested.

J9 relies heavily on standard file interfaces (`open`, `close`, `read`, `write`, `seek`, and `stat`), both for loading Java class files and for supporting the I/O services that the JVM provides to its client code. These interfaces are mapped directly to the 9P protocol, providing natural access to the filesystem exported by the 9P gateway server running on the controller partition. This filesystem may be the local filesystem of the controller, a network filesystem such as NFS or GPFS [15], or a combination of both kinds of filesystems.

The Linux port of J9 uses a subset of the Pthreads library [29] for threading services. Libra provides an implementation of that subset. Required services include thread creation and destruction, condition variables, and mutual-exclusion locks. The threading implementation was borrowed from the K42 operating system [26]. It is designed for scalability (with processor affinity and fully-distributed synchronization queues), although Libra does not currently support more than one processor per partition. So far we have found simple, round-robin scheduling (with no preemptive time-slicing) to be sufficient for the workloads we are using, but we expect to add time-slicing and more-sophisticated scheduling policies as options as the need arises.

To provide communication services to its client code, J9 makes calls on standard Linux socket interfaces (`bind`, `listen`, `accept`, and `connect`, as well as `send` and `recv`). In Libra, these calls are forwarded to the gateway server on the controlling partition, which in turn directs them (via standard 9P mechanisms) to a network server. The network server may be part of the gateway server itself, or may be running on another machine entirely. To the outside world, the Java program appears to be running on whatever system is running the network server. By running one network server for a cluster of separate machines, we can create the appearance (for management purposes) of a large number of Java programs all running under one internet address. There is a tradeoff here between network performance and management simplification. The appropriate degree of clustering will be different for different applications. For truly network-intensive applications, going through the controller partition may be too costly. We intend to explore partitionable devices and direct Libra socket support for such applications.

3.5 Limitations

The primary limitation of our current approach is that we made an explicit design decision to provide a new, separate name space for the functions exported by Libra (i.e., `libra_open` instead of `open`). The motivation for this decision was that it would allow us to be more flexible in the APIs we used to expose Libra functionality to J9. For the core J9 VM and native libraries this decision worked reasonably well, because the code was already vectoring through a port library layer and thus we could make the name space and API changes in a central location. However, all user written native methods (native methods not in `java.*`) would have to be ported/recompiled to use the Libra variant of any `libc` functions they relied on. After more experience with this approach, we now believe that Libra's API should include a subset of the `libc` API to enable easier porting. Doing this does not preclude adding additional non-standard functions that could be used by code that wants to exploit capabilities specific to the Libra environment.

One of the implications of the above decision is that the set of native methods we can execute is limited to those that have been ported to Libra. Therefore our current system does not support the full J2SE Java libraries. Also, the JVM normally uses `dlopen`[3] to load user-supplied native methods, and we have not implemented `dlopen` yet. Currently, only native methods that are statically linked into the J9 executable can be called.

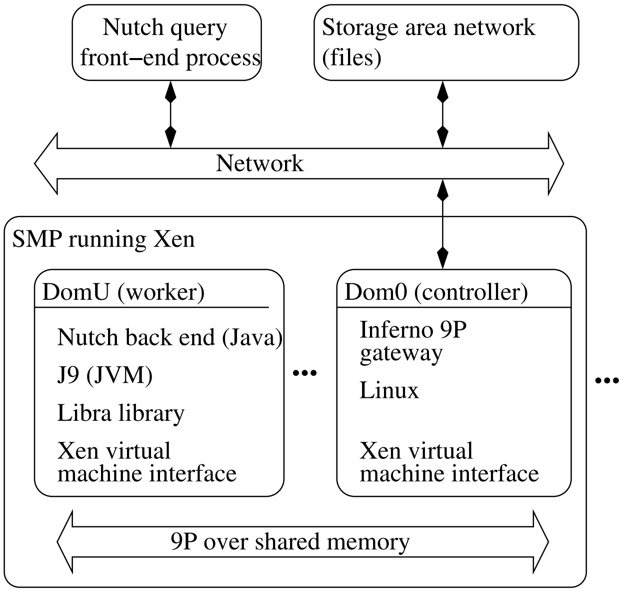

Figure 3. An overview of our Nutch application deployment.

A second limitation of our system is that it currently supports just one processor per Libra partition. This shortcoming is mainly attributable to the fact that support for multiprocessor secondary partitions was not yet available in Xen for PowerPC when this paper was written. Libra has been designed from the start with multiprocessor support and scalability in mind. Shared data structures are synchronized with scalable locks, and thread-scheduling data structures are maintained on a per-processor basis and are protected by disabling interrupts. The ability to cheaply disable and enable interrupts is one of the benefits of running in the privileged environment a hypervisor provides. The major issues still to be addressed for full multiprocessor support are booting/initialization and, of course, testing.

4. Nutch on J9/Libra

Nutch [8, 9] is an open-source web search application suite that uses the Lucene [17] search libraries. From among the various components of Nutch, we selected the query application as a test subject for Libra support and optimization.

4.1 Basic Configuration

The Nutch query application scans a database of web pages for a specified search pattern. (Other Nutch applications, the crawler and the indexer, build the database beforehand.) The application has two components, a front-end process that accepts queries from users via a web interface and formats the answers, and a set of back-end servers that do the actual searching. The full data set is partitioned among the back-end servers, so the front-end process broadcasts each query to all the servers. Each back end searches its part of the data set and returns to the front end a set of identifiers for its best-matching pages, along with a score for each match. The front-end process sorts the results by score, chooses the top

[3] `dlopen` is the programming interface to the dynamic linking loader.

n (where n is a parameter of the request), and then asks the back-end servers that have the best matches to return a “snippet” from each matching page. The snippets are displayed along with links to the original web pages. The front-end process can handle multiple simultaneous queries and keeps many concurrent requests to the back-end servers in flight.

We refer collectively to a front-end process and its back-end servers as a query “cluster”. For greater throughput, multiple clusters can be operated simultaneously. The clusters all search the same data set (which may be replicated or may reside in a filesystem designed for concurrent access), so a given query can be directed to any one of the front ends.

For our experiments, we concentrated on running the Nutch query back-end servers on J9/Libra. Figure 3 is an overview of the components involved in the J9/Libra incarnation of the Nutch distributed search application. The top of the figure shows the front-end process and the storage-area network that holds the data set. The bottom shows a machine that hosts one or more back-end servers and their controller. As Section 2 explained, the controller partition (dom0) runs Linux and hosts a gateway server for each back-end partition. Each back-end server runs in a Xen “domU” partition; these partitions run application Java code, which in this case is the Nutch query back-end server code, on the J9/Libra environment. The back-end servers communicate with the gateways over shared-memory channels set up when the back-end partitions are created.

4.2 Customizing Libra for Nutch

An advantage of Libra is that we can customize it to better support a specific workload. We implemented two optimizations in Libra, file-caching and socket-streaming, specifically to improve the performance of the Nutch back-end server, although the same optimizations have since been shown to help other applications as well. We also implemented a mechanism, file-cache dump/load, to shorten the time it takes to bring a new query cluster online. We describe the optimizations next; see Section 5.3 for an evaluation of their performance.

4.2.1 File-caching

When a Nutch web-page data set is partitioned so that it can be served by multiple back-end servers, an index file is created for each segment of the data set. The size of the index is roughly 10% of the size of the segment. The index data is heavily used as the server processes queries and scores the matches. The main data set is accessed only for the purpose of returning snippets of the best-matching pages. The back end has acceptable performance only if the entire index file as well as the more commonly requested snippets reside in memory rather than on disk. The Nutch back-end server does not maintain this data in its own space but instead relies on the underlying operating system to keep heavily-used file data resident. This behavior presents Libra with both a challenge and an opportunity. The challenge arises because with Libra the real filesystem runs on the dom0 Linux partition. File data may be cached there, but getting at it via the 9P protocol and the user-mode gateway server adds significantly to file-access latency. Libra has the opportunity to cache file data locally, transparently to both the JVM and the Java application. In this case, the cache implementation is almost trivial, because the data being cached for this application is known to be read-only. (The fact that a cache implementation can be chosen based on the characteristics of a particular application is one of the strengths of the library operating system approach.) Access to file data cached in Libra can be significantly faster than access to resident file data in Linux (or any general-purpose operating system) because system-call overhead is avoided.

4.2.2 Socket-streaming

The Nutch back-end server is a throughput engine. Data (search queries and snippet requests) arrives on a socket, and transformed data (matches and snippets) is returned on the socket. Because the front-end process keeps many requests in flight simultaneously, the arrival of new requests is only tenuously dependent on the answers generated for previous requests. The computational resources required by this application for pattern matching are significant enough that communication bandwidth should not be a bottleneck.

Profiling, however, shows that even with file-caching in place, the processor is idle for a significant fraction of the time. Here again, the problem is the latency of the 9P interactions with the gateway server, this time for the `recv` and `send` socket calls. The application makes a `recv` call only when it has run out of work to do. On Linux, the `recv` call is likely to return immediately with new request data that has already arrived and been buffered in the kernel, while Libra requires a 9P round trip to retrieve data from the gateway server’s host system. Similarly, when a back-end worker thread finishes a request and makes a `send` call to return an answer, the `send` call on Linux buffers the data in the kernel and returns immediately, while on Libra it again suffers the latency of a round trip to the gateway server.

To alleviate these problems, we implemented a socket streaming layer in Libra to decouple the application’s socket requests from the 9P requests to the gateway server. The streaming layer keeps a receive request posted to the gateway server at all times, so that incoming data is staged into the Libra partition before the application asks for it, and it buffers send requests and forwards them to the gateway server in batches. As with the file-caching optimization, the socket-streaming optimization is not appropriate for all applications. It improves throughput at some cost in latency (because of the extra layer), so the ability to customize the library for particular applications is important.

4.2.3 File-cache dump/load

For large-scale throughput applications such as Nutch query, it is often desirable to add and remove computing resources as demand shifts. For this reason, it is important to be able to bring a query cluster online quickly. One impediment to a quick start-up is the time it takes to bring the index file and commonly requested snippets into the file cache. Starting with a cold cache (on Libra or on Linux), it can take many minutes and thousands of queries to warm the cache to the point that the cluster can sustain a reasonable query rate.

On Libra we have the opportunity to shorten this process by cloning the file caches from an already-warmed-up cluster to a new cluster. Because the clusters all search the same data set, and because the new cluster will be joining in serving the same query stream, it’s highly likely that the cache content from the running cluster will be a good starting point for the new cluster.

We implemented a file-cache dump/load mechanism in Libra to experiment with this idea. A system administrator can send a request to a running cluster causing each of its back-end servers to dump the current content of its cache to a file, and the servers in the new cluster can then load the cache content from the filesystem when they start. The content is transferred in an orderly manner in very large blocks, so the process is much more efficient than the piecemeal warming of the caches that occurs when the servers start out cold. This mechanism, beneficial and yet invisible to the application, would be hard to implement in a general-purpose operating system. It is an example of a feature that is valuable for only a small class of applications and which should therefore be selectable when the library is configured for a particular use.

5. Performance

We report on three aspects of the performance of our system: microbenchmarks, standard Java benchmarks, and a Nutch web-search application that motivated some design and customization of our system.

We expect that for compute-bound workloads, Java or not, our system should have performance equal to or better than that of standard operating systems. For pure computation the performance should in fact be identical, except that on a standard system the workload may suffer interference (both in terms of processor cycles and cache pollution) from background operating system activity (daemons and interrupts) [18]. In a libOS environment, further optimizations driven by application-specific requirements are possible. For example, the application may benefit from dedicated memory resources under application control usage [19] and from appropriate thread-scheduling policies.

For workloads that make heavy use of system services, performance may be better or worse, depending on whether the services are ones that can be handled directly in the libOS or instead have to be forwarded to the controlling partition. Memory management and thread-switching operations are examples of the former. Filesystem or device access are examples of the latter, although customized support for important special cases is possible in a libOS. Filesystem data can be cached locally, providing faster access to heavily-used data than a general-purpose OS can provide, and direct access to some partitionable devices can be provided.

All experiments were run on IBM BladeCenter JS21s [22] with 2-socket dual-core PowerPC 970MP processors running at 2.5GHz, with 8GB RAM, and a 1MB L2 cache per core. Xen for PowerPC (XenPPC) [43] was used as the hypervisor, and tests were executed in hypervisor partitions that each had one physical core and 1920MB of memory. The memory size was chosen so as to allow 4 partitions simultaneously, with enough of the 8GB left over for the hypervisor itself.

In the following comparisons we use two versions of the Libra system, one that implements a local cache for read-only filesystem data (J9-Libra-fc) and one that does not (J9-Libra-nc). Socket-streaming is only enabled for experiments with the Nutch application. The same version of the J9 JVM is compiled for both Libra and Linux, and both use the same subset of the Java 1.4 J2SE libraries described earlier.[4] Linux measurements (J9-dom0) are made running the JVM on the controller Linux on the dom0 partition under Xen.

5.1 Microbenchmarks

Figure 4 shows throughput for Section 1 of the Java Grande Forum multi-threaded benchmarks [24] on J9-Libra-nc relative to the performance of the same on J9-dom0. File caching makes no difference for this benchmark, so results for only one of the Libra system configurations are shown. Libra provides all thread services needed by the JVM natively, while Linux requires a system call for at least some operations. J9-Libra clearly outperforms J9-dom0 on some of the tests (e.g. "Barrier:Simple"), while performing equivalently on the rest.

As stated previously, many system services are not handled directly by Libra. Instead, we rely on services provided by the operating system running in the control partition. Notable among these are filesystem and network I/O. Figure 5 illustrates the performance of forwarding standard filesystem read and write operations on a 128MB file with varying buffer sizes. The benchmark and workload are configured so that all data read should be present within the Linux page cache. Since in our design all remote resource actions devolve into filesystem operations, the difference between Libra and Linux-native I/O performance is representative of the overhead of most remote operations.

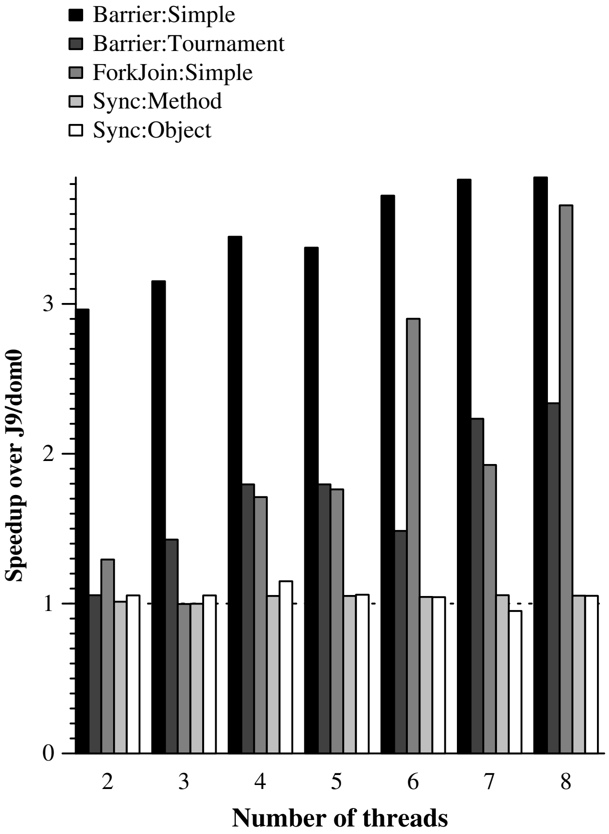

Figure 4. Speedup of J9/Libra over J9/dom0 on the Java Grande Forum Thread Benchmark Suite.

Raw read performance (without a Libra file-cache) varies from $1/12$ to $1/3$ that of native Linux due to the increased latency incurred by forwarding the operation. In our test environment, a read operation of 1024 bytes of cached data typically incurs around a 4 microsecond overhead on Linux. Accessing the same data from J9-Libra-nc incurs a 37 microsecond latency, decreasing performance by roughly an order of magnitude. As buffer sizes increase, the latency per message increases somewhat disproportionately on Libra due to a relatively naive transport implementation. However, since the total number of operations decrease, the relative performance of J9-Libra-nc improves.

When the Libra file cache is used, read performance improves dramatically. Since requests are satisfied completely within the Libra partition, there is no overhead for forwarding the operation. Furthermore, since the operation is handled entirely as a library call versus a system call, the latency of a read operation is about half that of native Linux. The native block-size of the J9-Libra-fc cache is 64KB, as such it also benefits in this particular benchmark from prefetching. The effect of both of these performance advantages decrease as the buffer size increases, resulting in roughly equivalent performance when using 64KB buffers.

Write operations are buffered by Linux, but eventually trigger disk operations which results in a higher average latency of approximately 12 microseconds. Libra currently makes no attempt to buffer write operations, so all actions are directly remoted with approximately the same per-operation overhead as the reads. However, since Linux write operations already incur higher latency the

[4] Because of the incomplete libraries, the SPECjvm98 and SPECjbb2000 results reported in this paper do not conform to SPEC run rules.

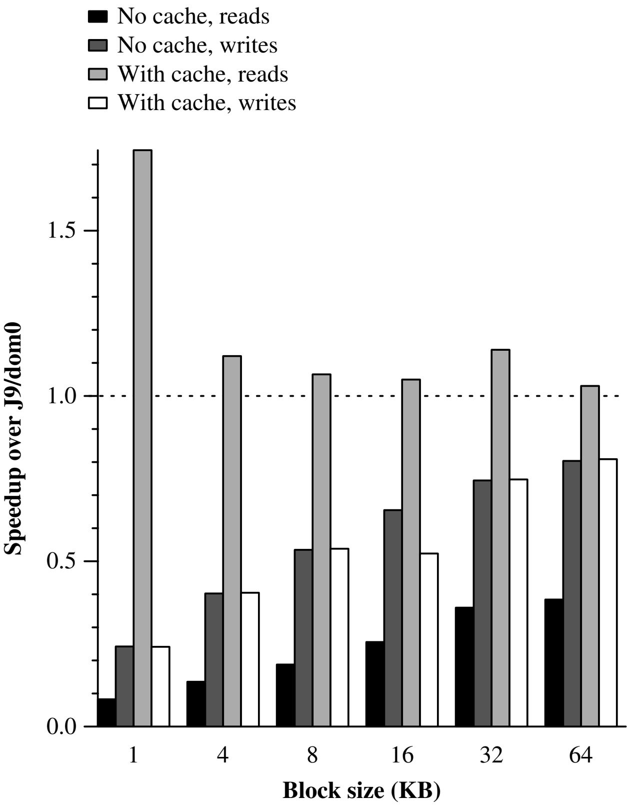

Figure 5. Speedups of file-block reads and writes: J9/Libra over J9/dom0, with and without a file cache.

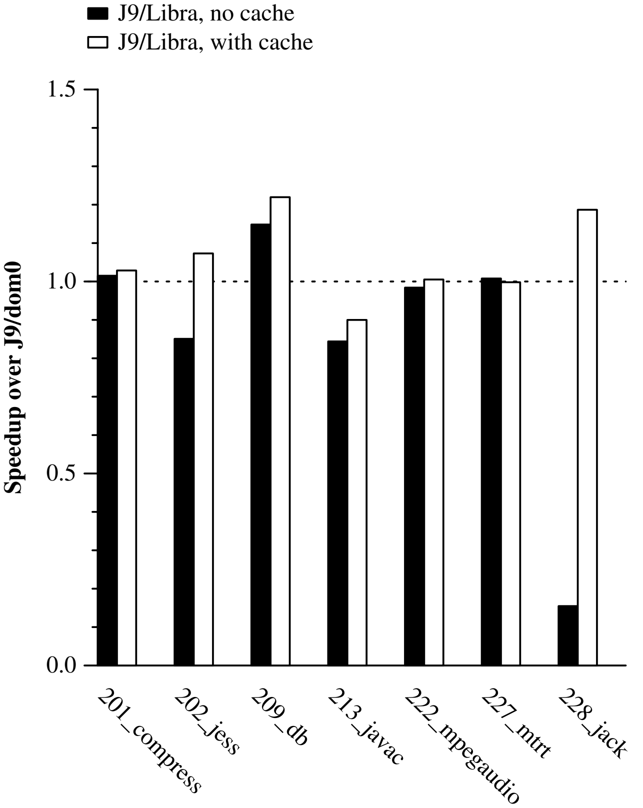

Figure 6. Speedups on the SPECjvm98 benchmarks: J9/Libra over J9/dom0, with and without a file cache.

Name	Time (s)		Reads	KB read
	Cache	No cache		
201_compress	6.00	6.08	25	45950
202_jess	1.68	2.12	12539	12
209_db	10.56	11.22	19048	1161
213_javac	3.38	3.60	4992	8261
222_mpegaudio	3.87	3.95	1956	3180
227_mtrt	2.07	2.05	344	678
228_jack	1.48	11.38	289901	283

Table 1. Effect of the file cache on SPECjvm98.

relative performance impact of writes from Libra is reduced significantly, varying between 1.2 to 4 times the overhead of a native write.

A similar effect can be observed for both read and write operations when the filesystem being accessed is mounted on the controller via NFS. In such an environment, write operations with buffer sizes of 4KB and higher are essentially identical. We believe that workloads with poor cache locality and other scenarios with high storage pressure would behave similarly.

5.2 JVM Benchmarks

The file-caching optimization was implemented in Libra specifically for the Nutch web-searching application (see Section 4), but we found it useful for the read-only input files of the SPECjvm98 benchmarks as well. Figure 6 shows the performance of the SPECjvm98 benchmarks on J9/Libra, with and without file-caching, relative to their performance on Linux on the controller partition under Xen. Each program was run in a separate JVM invocation. For each program we report the best time from 20 repetitions with input size 100, using the autorun mode (-s100 -m20 -M20 -a). The application and its data files reside on a local ext3 filesystem.

Three of the benchmarks (`compress`, `mpegaudio`, and `mtrt`) show equivalent performance under Linux and Libra (with or without file-caching). One (`db`) performs better under Libra and one (`javac`) performs worse, also independent of file-caching. For the remaining two (`jess` and `jack`), file-caching determines whether performance under Libra compares favorably with performance under Linux or not.

Table 1 helps explain why file-caching makes a real difference for some of the benchmarks and not for others. It shows the execution times of the different benchmarks under J9/Libra with and without file-caching, and it also shows the number of `read` calls made and the total number of kilobytes read by a single iteration of each application. We see that `jess` and `jack`, the two benchmarks for which caching is most beneficial, each make a very large number of small reads. The file cache helps these applications, not because it amortizes costs across multiple reads of the same data, but because it coalesces many small reads into a few large ones.

The `db` application also makes a large number of relatively small reads (62 bytes, on average), and file-caching indeed improves its performance. But `db` is compute-bound, and its performance is better under Libra than under Linux, even without file-caching, for a reason yet to be determined.

File-caching helps the performance of `javac`, but not enough to bring it up to the level it achieves under Linux. Preliminary indications are that in addition to `read` calls, it makes a large number of `open` and `stat` calls, the results of which are not cached. There may be an opportunity here for a more aggressive caching strategy.

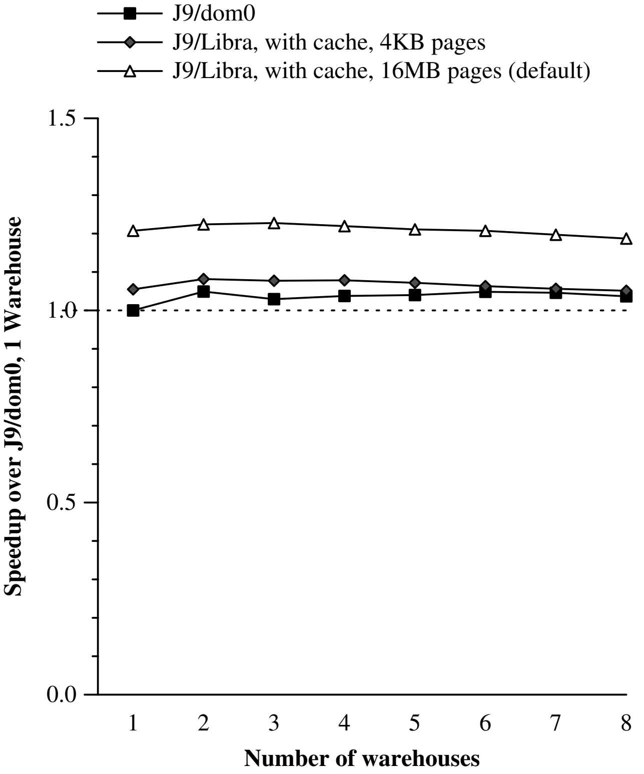

Figure 7. SPECjbb2000 throughput normalized to J9/dom0 at 1 warehouse. J9/Libra with both 16MB pages (default) and with 4KB pages.

Figure 7 shows the normalized throughput of the SPECjbb2000 benchmark running on Libra and on Linux, for from 1 to 8 "warehouses". The application was run on just one processor on both systems, so it is not surprising that the throughput does not increase with the number of warehouses. The file cache in Libra makes no noticeable difference for this benchmark, so cache-disabled results are not shown. The JVM heap size was fixed at 512MB for all cases.

Our initial measurements showed SPECjbb2000 performing consistently better (by 18% to 20%) on Libra than on Linux. More efficient threading and synchronization primitives might explain a small disparity, but nothing of that magnitude. It turns out SPECjbb2000 is known to benefit from the large virtual page sizes provided by some modern memory-management architectures. The PowerPC 970 processor supports two page sizes, the traditional 4KB page and a large 16MB page. Mapping large, flat regions with large pages reduces the pressure on translation lookaside buffers or other address-translation hardware structures, allowing the processor to spend more time computing and less time waiting for the memory-management unit to look up translations in page tables. Libra maps all its memory in one flat space, and it naturally uses the large page size to do it. Linux, on the other hand, uses 4KB pages by default (with good reason), and a program such as a JVM has to specifically request large pages if it wants them for parts of its address space. For experimental purposes, we changed Libra to use small rather than large pages. Figure 7 shows SPECjbb2000 performance on Libra for both page sizes. Performance with 4KB pages is only slightly better than that on Linux. It is possible that the comparatively good performance of the SPECjvm98 `db` benchmark on J9/Libra is also attributable to the use of large pages, but that guess remains to be verified.

Configuration	Queries per second
Default	5.9
File-caching	12.8
File-caching & socket-streaming	16.0

Table 2. Nutch queries per second on a single back-end server under three configurations: with no optimization, with file-caching, and with both file-caching and socket-streaming.

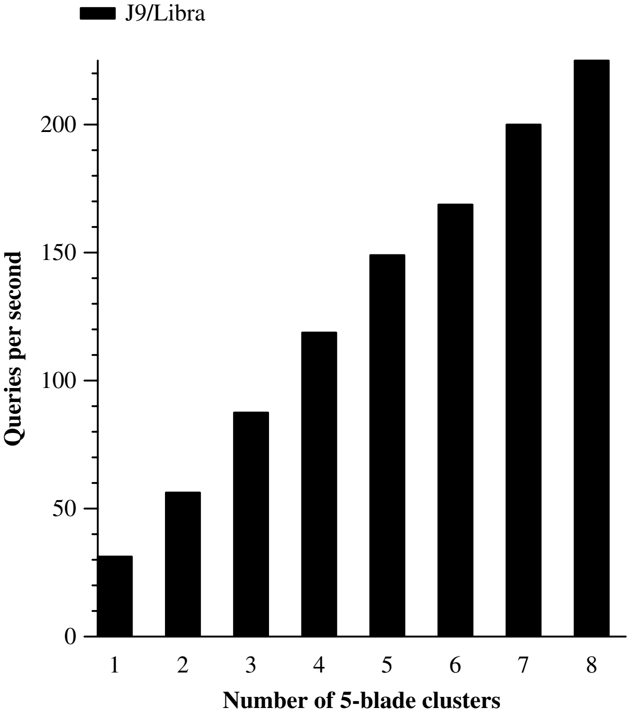

Figure 8. Nutch queries per second versus number of clusters serving requests. Each cluster has 5 4-core blades.

5.3 Nutch Benchmark

For the Nutch query application, the relevant performance metric is queries-per-second. We present query throughput results for a single back-end server running on Libra and for varying numbers of server clusters.

For our experiments, we arbitrarily chose a 150GB data set and partitioned it into 15 10GB segments. The index for each segment is roughly 1GB in size. The data sets and indices are housed in a Linux ext3 filesystem on a SAN-connected storage device.

Table 2 shows the query throughput for a single back-end server running on Libra and searching just one of the 10GB segments. The rows in the table correspond to different Libra configurations. The back end can process fewer than 6 queries per second with Libra configured without the file-caching and socket-streaming optimizations. Throughput more than doubles when file caching is enabled and improves by another 25% when socket streaming is added.

Our test machines have 4 cores each, so for our 15-segment cluster experiments, we used 5 machines for each cluster. Each machine runs one Linux/dom0 controller partition and 3 Libra/J9 back-end partitions, each searching one of the 15 segments of the partitioned data set. The front-end process for the cluster runs on one of the Linux/dom0 partitions. File-caching and socket-streaming are enabled.

Figure 8 shows total system throughput as 5-machine query clusters are added to the system. A single cluster, with 15 back-end servers jointly searching a 150GB data set, achieves about 31 queries per second. (This number can't be compared with the 16 queries per second that a single back end searching a 10GB segment achieves, because the ratio of snippet requests to search requests changes as more back-end servers are added.) The figure shows that throughput scales reasonably well as clusters are added. Eight clusters (40 machines in total) achieve a throughput of about 225 queries per second, for a speedup of more than 7 over a single 5-machine cluster.

We experimented with the same workload on Xen user domains (domUs) running Linux. Throughput on Linux/domU was roughly 20% lower than on Libra. However, these results are rather preliminary, as the Linux experiments were somewhat hampered by the immaturity of the underlying XenPPC hypervisor. We expect that with careful tuning of the application on Linux/domU, the performance gap will be reduced or closed. On the other hand, we also expect to find more opportunities for optimizing Libra for this application (see Section 7).

6. Related Work

BEA Systems has announced a version of its Java Virtual Machine named LiquidVM which runs directly on top of a VMware partition [10]. This will be used to run their custom WebLogic middleware application. LiquidVM has optimized TCP/IP and filesystem implementations that provide fast I/O but whose use incurs additional administrative overhead. The primary focus of LiquidVM seems to be on running general-purpose JVM workloads, with no provision for application-specific customization.

Project SPIN [7], VINO [35], and Scout [28] have explored the specialization of operating systems to meet the performance and functionality requirements of applications:

- In the SPIN operating system, the specialization is done through extensions written in a type-safe language that are dynamically linked into the operating system kernel. Applications can be written in any language and execute within their own virtual address space, but code that requires low-latency access to system services has to be written in the safe extension language.
- VINO supports the download of application extensions into the kernel. It uses software fault isolation to safeguard the kernel and a lightweight transaction system to cope with resource-hoarding.
- Scout is an operating system targeting network appliances, which typically perform a single specialized function. The specialized operating system behavior in Scout is achieved by configuring each Scout instance as a composition of available building-blocks. Scout is designed around a communication-oriented abstractions called *paths*; all resource-management decisions are made on a per-path basis.

Unlike these three systems, Libra does not bring specialized functionality into the kernel. Libra implements traditional operating system services directly into application-level libraries, as done in the Exokernel operating system [25]. Also, Libra differs from SPIN, VINO, Scout, and Exokernel on its goal of supporting specialized behavior of existing applications.

Synthesis [34] and Synthetix [33] provide improved flexibility and functionality by identifying commonly executed paths and producing optimized versions of them, on a per-system-call basis. In Libra we are able to optimize specific paths, but also alter higher-level services such as scheduling and memory management.

The Proxos system [41] allows applications to isolate their most sensitive components from the OS itself. The application developer specifies which system calls should be served by the untrusted commodity OS and which are to be handled by a trusted private OS. This partition of functionality focuses on security while our work aims to open up new opportunities for optimization of both existing and novel APIs.

Singularity [21] is a micro-kernel operating system for the construction of dependable systems. Singularity eliminates the distinction between an operating system and a safe language run-time system. The Singularity architecture can host completely different run-time systems, allowing each process to have its own customized version. For example, processes with certain allocation strategies may be able to pre-allocate or stack-allocate memory for all used objects, obviating the need for a garbage collector in the runtime. Like SPIN, VINO, and Scout, Singularity is not appropriate for existing systems.

In the High Performance Computing arena, the Blue Gene/L system is an example of deploying a specialized kernel to directly support applications. Nodes are split into two types: the I/O nodes, which run the Linux operating system, and compute nodes which run a custom kernel called the Blue Gene/L Run Time Supervisor (BLRTS) [1]. The BLRTS is extremely lightweight, co-exists in the same address space as the application, and remotes many system calls to a specialized daemon running on the I/O Node, which provides job control and I/O operations. BLRTS addresses the requirements of a large class of applications, but it does not aim at application-specific optimizations.

Specialized hardware has been used to accelerate Java applications. Azul Systems has designed a custom system (CPU, chip, board, and OS) specifically to run garbage collected virtual machines [11]. In 2003, IBM designed and developed the System z Application Assist Processor (zAAP) for its System z9 mainframe line [13]. While the underlying hardware for the processor is identical to the general-purpose z9 processors, the zAAP's microcode is optimized for the execution of Java workloads. The zAAP relies on more conventionally configured processors and the general-purpose z/OS operating system to function. Similarly, the Integrated Facility for Linux (IFL) [45] and Z Integrated Information Processors (zIIP) [44] function as offload processors for Linux and DB2. All three represent microcode-implemented optimizations for hardware. In the case of IFL, the Linux systems run on logical partitions on top of the specialized processors, and use HiperSockets [6] to communicate with each other as well as other processors on the mainframe.

7. Future Work

In the short-term, there are some limitations of the initial implementation that can be addressed with straightforward engineering.

The main functional limitation is the inability to support arbitrary native methods. By moving the Libra API to be a subset of the standard `libc` API, we will be able to run many native methods in the Libra partition. Handling native methods that stray outside of this subset could be implemented either lazily by redirecting to the controller partition when a native method actually calls an unsupported function or eagerly by analyzing the method when it is dynamically loaded and if necessary marking it to be redirected when invoked.

Much of the overhead in our remote I/O model is due to multiple unnecessary copies of data and header information present in our current transport implementation. User-space gateway servers on the controller add further copies and latency. Most of these copies and associated latency can be removed by a rework of the underlying transport and integration of the gateway services into the controller kernel. Providing facilities for direct read-only memory sharing and shared page-caches would provide further performance and efficiency gains, particularly on systems running multiple application partitions.

Longer term, we believe that our approach opens up a number of interesting avenues for future research.

We believe that our architecture enables many JVM optimizations beyond those we describe in this paper. For example, in JVMs that use "safe-points" to support type-accurate garbage collection, whenever the JVM wants to initiate a garbage collection cycle it must roll forward all runnable threads to their next safe point. If instead the JVM and the operating system cooperated to ensure that thread context–switches only occurred at JVM-level safe points, the transition to garbage collection could be accomplished instantly. Although rolling forward is not a significant cost in stop-the-world collectors, it can impact the performance of incremental collectors such as Metronome [3]. Another promising possibility is that because the JVM could handle interrupts directly instead of receiving them second-hand from the OS, read barriers implemented with the virtual-memory hardware could be faster than software read barriers [11]. In general, the JVM could rely on our system to provide stronger invariants than traditional systems provide and optimize accordingly.

Real-time applications may also benefit from our architecture. Modern hypervisors allow partitions to run in real mode [2]; together with advances in real-time garbage collection [3, 23] and deterministic Java tasks (eventrons) [36], a system like ours could run hard real-time tasks and even device drivers written in Java. Legacy code could run in other partitions without breaking the real-time guarantees.

As processors acquire more cores and clusters acquire more processors, operating systems will need to scale beyond small SMPs. We believe that hypervisors are an ideal platform for such systems.

8. Conclusions

In this paper we have described our vision of how hypervisors can be used to transform existing software systems into high-performance exokernel systems. Our progress to date in building Libra, a library operating system specialized for running particular classes of workloads on the J9 JVM, leads us to believe that this is a viable approach for optimizing the performance of important workloads and subsystems. We intend to continue to improve Libra and to explore the extent to which the co-design of the libOS and the JVM can yield significant performance or functional advantages for large-scale Java applications that could not easily be obtained in traditional execution environments.

However, engineering custom operating system components for each application or service is not tractable in the long term. Many possible customizations can be organized into recognizable components with well-defined interfaces which may be interchanged by the developer or even during execution. Some of the key questions to be explored in such an effort are:

- What are the defining characteristics of execution environments and to what extent are they architecture dependent?
- Can abstractions within customized environments be standardized to an extent that maintains the portability of applications, or do standards stifle innovation?
- What are the right tools and languages for implementing a component systems infrastructure?
- What degree of dynamic customization needs to be supported?

We believe that further research along these lines could dramatically alter the way in which software stacks are developed and deployed.

Acknowledgments

We thank David Bacon, Muli Ben-Yehuda, Mark Mergen, Michal Ostrowski, Volkmar Uhlig, Amos Waterland, and Jimi Xenidis for their valuable insights.

This material is based upon work supported in part by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

References

[1] G. Almási, R. Bellofatto, J. Brunheroto, C. Caşcaval, J. G. Castaños, L. Ceze, P. Crumley, C. C. Erway, J. Gagliano, D. Lieber, X. Martorell, J. Moreira, A. Sanomiya, and K. Strauss. An overview of the Blue Gene/L system software organization. In *Proceedings of the Euro-Par Conference on Parallel and Distributed Computing*, 2003.

[2] W. J. Armstrong, R. L. Arndt, D. C. Boutcher, R. G. Kovacs, D. Larson, K. A. Lucke, N. Nayar, and R. C. Swanberg. Advanced virtualization capabilities of POWER5 systems. *IBM Journal of Research and Development*, 49(4):523–540, 2005.

[3] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 285–298, Jan. 2003.

[4] C. Bailey. Java technology, IBM style: Introduction to the IBM developer kit. `http://www-128.ibm.com/developerworks/java/library/j-ibmjava1.html`, May 2006.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating System Principles*, Bolton Landing, New York, U.S.A., 2003.

[6] M. Baskey, M. Eder, D. Elko, B. Ratcliff, and D. Schmidt. zSeries features for optimized sockets-based messaging: Hipersockets and OSA-Express. *IBM Journal of Research and Development*, 46(4/5), April 2002.

[7] 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 *SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating systems Principles*, pages 267–283. ACM Press, 1995.

[8] M. Cafarella and D. Cutting. Building Nutch: Open source search. *Queue*, 2(2):54–61, 2004.

[9] M. J. Cafarella and O. Etzioni. A search engine for natural language applications. In *WWW '05: Proceedings of the 14th International World Wide Web Conference*, pages 442–452. ACM Press, 2005.

[10] G. Clarke. BEA adopts virtual strategy with VMware. *The Register*, December 2006.

[11] C. Click, G. Tene, and M. Wolf. The pauseless GC algorithm. In *VEE '05: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, pages 46–56. ACM Press, 2005.

[12] S. Crosby and D. Brown. The virtualization reality. *Queue*, 4(10):34–41, 2007.

[13] D. Deese. Introduction to zSeries Application Assist Processor (zAAP). In *Proceedings of the 32nd International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems*, pages Vol 2, 517–528. Computer Measurement Group, 2005.

[14] R. Figueiredo, P. A. Dinda, and J. Fortes. Resource virtualization renaissance. *IEEE Computer*, 38(5):28–69, 2005.

[15] General parallel file system (GPFS). `http://www.almaden.ibm.com/StorageSystems/File_Systems/GPFS/`.

[16] G. Guardiola, R. Cox, and E. V. Hensbergen. Persistent 9P sessions for Plan 9. In *Proceedings of 1st International Workshop on Plan 9*, December 2006.

[17] E. Hatcher and O. Gospodnetic. *Lucene in Action*. Manning Publications, 2004.

[18] E. V. Hensbergen. The effect of virtualization on OS interference. In *Proceedings of the 1st Annual Workshop on Operating System Interference in High Performance Applications*, August 2005.

[19] E. V. Hensbergen. Partitioned reliable operating system environment. *Operating Systems Review*, 40(2), April 2006.

[20] E. V. Hensbergen and K. Goss. PROSE I/O. In *Proceedings of 1st International Workshop on Plan 9*, December 2006.

[21] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[22] IBM Corporation. BladeCenter JS21. `http://www-03.ibm.com/systems/bladecenter/js21/`.

[23] IBM Corporation. *WebSphere Real-Time User's Guide*, 2006.

[24] Java Grande Forum. Java Grande Forum Benchmark Suite. `http://www2.epcc.ed.ac.uk/javagrande/index_1.html`.

[25] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *SOSP '97: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 52–65, 1997.

[26] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a complete operating system. In *Proceedings of EuroSys'2006*, pages 133–145. ACM SIGOPS, April 2006.

[27] Linux: KVM paravirtualization. `http://kerneltrap.org/node/7545`, January 2007.

[28] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Symposium on Operating Systems Design and Implementation*, volume 30, pages 153–167. ACM, 1996.

[29] The Open Group Base Specifications Issue 6, IEEE Std 1003.1. `http://www.opengroup.org/onlinepubs/009695399/`, 2004.

[30] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[31] R. Pike, D. Presotto, S. Dorward, D. M. Ritchie, H. Trickey, and P. Winterbottom. The Inferno operating system. *Bell Labs Technical Journal*, 2(1), Winter 1997.

[32] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in Plan 9. In *Proceedings of the 5th ACM SIGOPS Workshop*, Mont Saint-Michel, 1992.

[33] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: streamlining a commercial operating system. In *ACM Symposium on Operating System Principles*, volume 29, 3–6 December 1995.

[34] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[35] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. An introduction to the architecture of the VINO kernel. Technical report, Harvard University, 1994.

[36] D. Spoonhower, J. Auerbach, D. F. Bacon, P. Cheng, and D. Grove. Eventrons: a safe programming construct for high-frequency hard real-time applications. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294, 2006.

[37] R. M. Stallman, R. Pesch, and S. Shebs. *Debugging with GDB: The GNU Source-Level Debugger*. January 2002.

[38] Standard Performance Evaluation Corporation. SPECjbb2000 Java Business Benchmark. `http://www.spec.org/jbb2000`.

[39] Standard Performance Evaluation Corporation. SPECjvm98 Benchmarks. `http://www.spec.org/jvm98`.

[40] Sun MicroSystems. Connection limited device configuration; JSR 30, JSR 139. `http://java.sun.com/javame/reference/apis.jsp`.

[41] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 16th USENIX Symposium on Operating System design and Implementation*, pages 279–292, November 2006.

[42] B. Walters. VMware virtual platform. *Linux J.*, 1999(63es):6, 1999.

[43] XenSource. XenPPC. `http://wiki.xensource.com/xenwiki/XenPPC`.

[44] System z9 integrated information processor (zIIP). `http://www.ibm.com/systems/z/ziip`, 2006.

[45] Integrated facility for Linux. `http://www.ibm.com/systems/z/os/linux/ifl.html`, 2006.