

The Virtualized Virtual Machine: The Next Generation of Virtual Machine Technology

David F. Bacon
IBM T.J. Watson Research Center

ABSTRACT

Virtual machine technology will continue to increase in importance, eventually spanning from motes to supercomputers. In this position paper, I describe my personal vision for the virtual machine of the future: a virtual machine that dynamically modifies all portions of itself, rather than just the compiled code, and that dynamically re-configures the underlying hardware as well.

1. INTRODUCTION

Virtual machines have been enormously successful in the last decade, and virtually all new important languages now use virtual machine technology, in particular Java, C#, perl, and python.

Virtual machines have evolved from pure bytecode interpreters into systems with sophisticated dynamic compilers driven by profile feedback. However, while there has been enormous progress in the area, I believe the development of virtual machines is still in its infancy.

In this paper I will discuss three major areas in which I believe virtual machines will evolve in the coming decade.

The first is in the extension of the dynamism that is now present in the compiled code into the run-time system itself, so that the basic structure of the virtual machine is itself dynamically re-configured to meet the needs to the running code.

The second is the extension of dynamism into the hardware, so that after optimized JIT compilation the next step in speeding up a hot method is to dynamically generate new instructions for it or compile whole loops or methods into programmable logic substrates.

The third is the extension of dynamism into the scheduler, so that code with very stringent scheduling requirements, like hard real-time tasks and interrupt handlers, can be written in a language like Java.

This paper makes no attempt at being thorough in terms of the areas covered (or in citation of relevant related work, of which there is much). Instead I present a purely personal view of the virtual machine landscape, discussing the work my colleagues and I have done which has partly been driven by, and partly drives, this vision of the virtual machine of the future.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Workshop on Language and Runtimes, LaR'04, October 2004, Vancouver, British Columbia.

Copyright © 2004 ACM \$5.00.

2. VIRTUALIZING THE RUN-TIME

The rise of virtual machines has seen the emergence of steadily more advanced technology for dynamic compilation [11, 13, 9, 15, 2, 16]. However, this compilation has always been performed within the context of a static virtual machine run-time system.

As the level of abstraction has increased, so has our ability to make changes in implementation without perturbing the application programs running above the virtual machine interface. At the same time, aggressive optimization of virtual machines, particularly for Java, has led to a steadily increasing modularization and parameterization of the run-time system itself.

Until now this parameterization has primarily been used to study implementation trade-offs and to statically build different variants of a virtual machine for different users.

The confluence of a three factors: increased level of abstraction, organized parameterization, and continued demand for performance will drive the development of virtual machines with are themselves adaptive: the entire run-time system, and the associated dynamic compilation, will be made dynamic to adapt to the changing and unpredictable needs of applications.

We now consider in concrete terms how the various components of the virtual machine could be made dynamic.

2.1 Garbage Collector

Since the acceptance of Java as a mainstream commercial programming language, there has been immense pressure to improve garbage collection technology. This has led to a large body of work in this area, and most virtual machines now include some sort of garbage collection framework [8] that allows different garbage collectors to be implemented and benchmarked against each other.

Until now the almost universal approach has been to seek a single “best” garbage collector, benchmarking a few collectors against each other over a set of benchmarks and choosing the one with the best overall performance.

However, recent work [17] has shown the efficacy of switching collectors dynamically in response to the workload (memory usage, allocation rate, or object lifetimes) or to the available resources (competition for physical memory by other processes, need to reduce power consumption). By using the generic collector framework, multiple collectors are simultaneously supported within a single virtual machine.

When the need to switch collectors is detected, a special-purpose garbage collection is performed in which the heap is re-formatted using the memory layout of the new collector. Changing garbage collectors is only modestly more expensive than performing a normal collection.

Making the collector dynamic adds a level of indirection, most notably in the memory allocation routines, which are extremely

performance-critical. However, adaptive compilation technology allows for the optimization of only a small number of methods, and when a collector switch is performed these methods can be re-compiled as needed.

In some cases, as when switching from a generational to a non-generational collector, the old optimized methods can be replaced at leisure, since the presence of write barriers will slow down the code slightly but not affect correctness. When performing the opposite transition, however, hot methods must be recompiled before they are executed again. This includes methods on the stack, so on-stack replacement is a necessary enabling technology for high performance collector switching.

The primary remaining challenges for this work are in developing the best strategy for determining the need to switch collectors. Unlike dynamic compilation, which can make use of fairly simple localized profiling, garbage collection is by its nature a global operation, so the necessary profiling is more difficult.

Approaches under exploration include offline profiling, dynamic selection based on training periods with different collectors, and selection based on measurement of application parameters such as allocation rate, live heap ratio, and object lifetime and connectivity.

2.1.1 *Dynamic Collector Generation*

In recent work [6], we showed how almost any garbage collector (tracing or reference counting) could be encompassed in a single framework. An enormous variety of collectors can be expressed in this framework, and the orthogonalization allows the conception of new collector organizations and algorithms heretofore unrealized.

Such a framework is a first step to an evolution beyond the dynamic selection from a statically fixed set of collectors to the dynamic generation of the collector itself. In such a system, the application would be measured and the memory would be divided into various regions, some of them independently collected, based on the measured properties of the total system.

The result is just-in-time generation of the collector itself. Such an approach will be especially valuable in a virtual machine that executes many processes, some of them tightly integrated and some of them barely related at all. Such cases will arise in virtual machines running systems like WebSphere, which contain many different servlets all in a single virtual machine image, or in virtual machine-based operating systems.

2.2 Object Model

The object model — that is, the way in which objects are represented in memory — has also been the subject of modification, variation, and parameterization. Some of this work relates to the fixed portion of the objects (class pointer, collector metadata, synchronization state, object hashcodes [7, 4]). Other work has focused on varying the representation of the data fields, generally to reduce object size [1, 10].

Just as with garbage collectors, there has been a trend to statically parameterize the object model. However, the object model can also be varied dynamically. This is both enabled by dynamic garbage collection (since it allows objects to be reformatted during collection) and demanded by it (since different collectors preferentially use different object layouts).

However, there are many other uses for dynamically adaptive object layout. The lock implementation of objects can be changed depending on whether the objects are never locked, are locked without contention, or are locked with heavy contention. Objects that are frequently hashed can have a high-resolution hash code included in the object at allocation time.

Small, numerous objects (such as complex numbers) can be allo-

cated without any inline metadata at all by placing them in special pages of memory, with the metadata (including the class pointer) instead being associated with the page.

Adaptation can also be applied to the data portion of objects. Fields that require fewer bits of precision than their data type declarations can be compressed; such compression could even be dynamic and adaptive. Similarly, some fields can be speculatively assumed to be constant.

Infrequently accessed objects can be compressed, or moved to slower memory (like flash), or paged out entirely.

2.3 Scheduling and Synchronization

Different applications have different types of scheduling requirements and properties. Some applications make extensive use of operating system facilities that require a one-to-one mapping between language-level and operating system-level threads. On the other hand, some applications consist mostly of computational threads with no operating system interaction whatsoever. Such threads can be scheduled at user-level, saving context switch costs and increasing determinism.

Some applications, especially for real-time systems, have very specific and exacting scheduling requirements. They may require reactivity at a much higher resolution than the typical 10 millisecond scheduler interrupt.

In fact, some applications can actually have their schedule compiled directly into the code [12]. In this sense we can consider the traditional dynamic scheduler as an interpreter, which is then “compiled out” when optimized code is generated.

3. VIRTUALIZED INSTRUCTION SET

Virtual machines also provide a new level of flexibility at the hardware level. Once a substantial amount of the code running on a modern processor is under the control of a virtual machine, the hardware itself can be made more malleable.

I argue that we are on the verge of a third major phase in computer instruction set architecture: the first was programmer-oriented, with instruction sets geared towards programmers who were using them directly. This approach reached its apotheosis in the Intel iAPX 432.

The second phase was compiler-oriented architecture, which began with the RISC movement: as fewer and fewer humans wrote machine language code, it made more sense to develop an ISA geared towards compilers rather than humans.

As the amount of code executed by virtual machines begins to dominate, we will see the rise of virtual machine-oriented architecture: instruction sets geared to dynamic compilers for languages with no fixed storage layout that can modify code and move objects at will.

3.1 Breaking the ISA Stranglehold

The hardware flexibility can manifest itself in a number of ways. First of all, it frees us from the tyranny of instruction set architecture (ISA) compatibility. Instructions can be provided for the sole use of dynamic compilers, with no need to guarantee their inclusion in future versions of the architecture. Why? Because the instructions exist only to support the virtual machine, and the virtual machine interface does not change, and the virtual machine can dynamically detect the availability of the instructions.

Since the introduction of the ISA concept with the IBM System/360 40 years ago, architects have been very loth to add new instructions, since history has shown that they may have to be supported for decades. In such an environment, a highly conservative approach makes sense.

However, such conservatism also stifles innovation. With the ability to add and remove instructions as technology trade-offs vary and as new techniques are developed, we can look forward to a renaissance of computer architecture driven by the close interaction between the developers of virtual machines and the developers of the chips that run them.

3.2 Variable Memory Formats

With languages like C, the layout of memory was part of the language specification. The fields of a `struct` had to be laid out in textual order, and each field had to occupy a given number of bits using a given representation.

A virtual machine with a fully abstract data model, like Java, allows us to change the way in which objects are represented in memory. Thus not only can we change the instructions, but we can add instructions that support variable bit-width operands, compressed operands, and so on. For instance, we could provide hardware support for representing Unicode characters with a single byte, and consulting a lookaside table for 16-bit characters.

3.3 Dynamically Generated Instructions

The most exciting possibility is for the dynamic generation of new instructions at run-time. This is the logical extension of the JIT principle. If a routine is optimized to the limit of the compiler's ability, and still consumes large amounts of time or space, an adaptive compiler could, with suitable hardware support, generate new instructions to further optimize the hot method.

Such optimization could take place at a micro or macro level. At the micro level, consider a program that performs many independent operations on signed 8-bit integers. The compiler could notice this and dynamically reconfigure the ALU to add an instruction that uses the existing adder circuitry to perform 4 8-bit adds at once in a single 32-bit register. This would allow dynamic generation of operations such as those found in the MMX instruction set.

At the macro level, systolic operations like packet filtering could be compiled onto an FPGA substrate included on the CPU chip. Such compilation is already performed on an application-specific basis in many domains.

In order to enable such compilation into hardware, it is necessary for the gap between the hardware and software description to narrow. We have developed a language called Kava [3], a backward-compatible extension to Java, in which primitive data types like `int` are eliminated. Instead, all data types are described in a bottom-up fashion out of primitive enumerations and fixed-size arrays.

For instance, an integer is an object containing a fixed-size array of 32 objects of type `bit`. From the programmer's point of view, the type is fully abstract: the `bit` type is a class like any other, and includes methods, static data, and so on. However, the type is defined in such a way that it can in fact be represented inside the machine with only one bit.

Kava thereby combines the representation-independent high-level data abstraction of Java with the bit-level specification of hardware description languages like VHDL and Verilog. By narrowing the semantic gap between hardware and software, it enables dynamic synthesis of hardware at run-time.

The simplest way of using this kind of abstraction is to provide high level data types which directly map into special-purpose instructions like MMX. For instance, a data type containing multiple values and an operator that does pair-wise addition, multiplication, and so on. The JIT compiler can then query the hardware for the availability of the special-purpose instructions, and if present generate them directly; if not, the data types are broken into individual "primitive" types (bytes, integers, floats, etc.) and processed with

the standard instruction set. In this scenario, the compiler is simply doing idiom recognition for a fixed set of special-purpose types.

The next level is to provide reconfigurable ALUs that can perform operations under various masks and use registers as a source of short-vector SIMD parallelism. Some examples are packing an aligned pointer and a data field (for instance tag bits) into a single word, or 3 10-bit integers and a 2-bit tag, and so on. With support for the appropriate mask, shift, sign-extension, and other operations, significant improvements in code density and execution time could be achieved.

Most ambitiously, a larger piece of code representing some high-level function, like a packet filter, could be turned into a function operating on fixed-size data by strip-mining loops, and then the resulting code could be translated into VHDL or Verilog and from there into an FPGA compiler, with the result being used to dynamically reconfigure a programmable hardware fabric adjacent to the CPU core.

4. REAL-TIME SYSTEMS

Real-time systems are another area where high-level language virtual machines have been notably absent. This is due primarily to the extreme variation in response that can be caused by garbage collection, but also by dynamic loading, just-in-time compilation, and other dynamic run-time behavior.

Our recent work on the Metronome garbage collector [5] has shown that hard real-time behavior is achievable on uniprocessors at a resolution of 5 milliseconds. Hard real-time means that when the target response time is 5 milliseconds, there are 0 intervals that fail to meet the target.

We are currently working on a second-generation system with the goal of driving latency down to a few hundred microseconds, while increasing available CPU capacity. For interrupt handlers and regular periodic tasks, we expect to be able to reach an accuracy in the neighborhood of 10 microseconds.

Real-time systems comprise an ever-growing portion of the total deployed processors, and the inability to program them in a general-purpose, high-level language has been a serious hindrance to the timely development of reliable products.

The ability to drive latencies well into the sub-millisecond regime opens up an enormous new range of applications. This time scale is well below human perception, and faster than many sensors and actuators can respond.

A major open problem is real-time garbage collection on multiprocessors. This will be a very active area of research in the coming decade. The possibility of providing specialized hardware support that provides the kind of low-cost atomicity required may speed the solution of this problem.

4.1 The Virtual Machine Operating System

Operating systems are one specialized type of real-time system.

In the early seventies, the prevailing wisdom was that operating systems had to be written in assembly language, because they were so performance critical. We are currently in a similar situation with respect to languages like Java and C#.

However, in the next five to ten years we will see the arrival of the first operating systems written entirely in high-level, virtual-machine based, garbage collected languages. A project at Microsoft Research call Singularity is beginning to investigate such an approach using C#.

The reason that operating systems are so demanding is that they require efficiency, explicit control, and precise timing.

The ability to write real-time code such as interrupt handlers, device drivers, and so on is traditionally limited by the use of garbage

collection in virtual machine languages. However, the real-time garbage collection technology we have just described can lift this barrier.

Meanwhile, improvements in JIT technology continue to narrow the gap between the performance of Java and C++.

While Java consciously removed the ability to control storage explicitly, its draconian enforcement simply pushed programmers into native C code, so the inflexibility was clearly a mistake. To some extent C# has rectified this by providing low-level, unsafe operations. For Java, low-level programming has been performed with ad-hoc interfaces like the `VM_Magic` class of Jikes RVM [14]. Inter-operability is largely an engineering issue, and over time the seams between a Java-like language and a C-like language will become smoother.

5. CONCLUSIONS

The shift from statically compiled languages to high-level virtual machines with dynamic compilation has opened up vast opportunities for innovation in the design and implementation both of the hardware and software that runs the programs. I believe we have only begun to scratch the surface, and that in the coming decade the flexibility provided by virtual machines bear fruit in many new and exciting ways.

6. REFERENCES

- [1] ANANIAN, C. S., AND RINARD, M. Data size optimizations for Java programs. In *Proceedings of the ACM SIGPLAN conference on Languages, Compilers, and tools for Embedded Systems* (San Diego, California, 2003), pp. 59–68.
- [2] ARNOLD, M., FINK, S., GROVE, D., M.HIND, AND SWEENEY, P. Adaptive optimization in the Jalapeño JVM. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Oct. 2000), pp. 47–65.
- [3] BACON, D. F. Kava: A Java dialect with a uniform object model for lightweight classes. *Concurrency—Practice and Experience* 15, 3–5 (Mar.–Apr. 2003), 185–206.
- [4] BACON, D. F., CHENG, P., AND GROVE, D. Garbage collection for embedded systems. In *Proceedings of the Fourth ACM International Conference on Embedded Software (EMSOFT)* (Pisa, Italy, Sept. 2004).
- [5] BACON, D. F., CHENG, P., AND RAJAN, V. T. 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* (New Orleans, Louisiana, Jan. 2003). *SIGPLAN Notices*, 38, 1, 285–298.
- [6] BACON, D. F., CHENG, P., AND RAJAN, V. T. A unified theory of garbage collection. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications* (Vancouver, British Columbia, Oct. 2004).
- [7] BACON, D. F., FINK, S. J., AND GROVE, D. Space- and time-efficient implementation of the Java object model. In *Proceedings of the Sixteenth European Conference on Object-Oriented Programming* (Málaga, Spain, June 2002), B. Magnusson, Ed., vol. 2374 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 111–132.
- [8] BLACKBURN, S., CHENG, P., AND MCKINLEY, K. Oil and water: High performance garbage collection in Java with MMTk. In *The 26th International Conference on Software Engineering* (Edinburg, Scotland, May 2004).
- [9] CHAMBERS, C., AND UNGAR, D. Making pure object-oriented languages practical. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Nov. 1991), pp. 1–15.
- [10] CHEN, G., KANDEMIR, M., VIJAYKRISHNAN, N., IRWIN, M. J., MATHISKE, B., AND WOLCZKO, M. Heap compression for memory-constrained Java environments. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Anaheim, California, Oct. 2003). *SIGPLAN Notices*, 38, 11, 282–301.
- [11] DEUTSCH, L. P., AND SCHIFFMAN, A. M. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Jan. 1984), pp. 297–302.
- [12] HENZINGER, T. A., KIRSCH, C. M., AND MATIC, S. Schedule-carrying code. In *Proc. of the Third International Conference on Embedded Software* (Philadelphia, Pennsylvania, Oct. 2003), R. Alur and I. Lee, Eds., vol. 2855 of *Lecture Notes in Computer Science*, pp. 241–256.
- [13] HÖLZLE, U., AND UNGAR, D. A third generation SELF implementation: Reconciling responsiveness with performance. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (1994), pp. 229–243.
- [14] IBM CORPORATION. *The Jikes Research Virtual Machine User Guide*, 2004.
- [15] ISHIZAKI, K., KAWAHITO, M., YASUE, T., TAKEUCHI, M., OGASAWARA, T., SUGANAMA, T., ONODERA, T., KOMATSU, H., AND NAKATANI, T. Design, implementation, and evaluation of optimizations in a Just-In-Time compiler. In *Proc. of the ACM 1999 Conference on Java Grande* (San Francisco, CA, June 1999), pp. 119–128.
- [16] PALECZNY, M., VICK, C., AND CLICK, C. The Java Hotspot Server Compiler. In *USENIX Java Virtual Machine Research and Technology Symposium* (Apr 2001), pp. 1 – 12.
- [17] SOMAN, S., KRINTZ, C., AND BACON, D. F. Dynamic selection of application-specific garbage collectors. Submitted for publication, 2004.