JikesNODE and PearColator: A Jikes RVM Operating System and Legacy Code Execution Environment

Dr. Ian Rogers, Dr. Chris Kirkham
The University of Manchester
{ian.rogers, christopher.kirkham}@manchester.ac.uk

Abstract

JikesNODE is an effort to create a Java operating system integrating the open source projects of the Jikes Research Virtual Machine (RVM) and the Java New Operating System Design Effort (JNODE), respectively a Java Virtual Machine (JVM) and Java operating system. PearColator is a novel use of the Jikes RVM to execute not its intended Java bytecodes, but instead PowerPC binaries. Our endeavours are lead by the desire for an architecture neutral system that exposes new opportunities for dynamic optimisations. This paper presents the structure of our system, JikesNODE and its bootstrapping process, and the PearColator legacy code execution environment.

1 Introduction

The Jamaica project is an effort to investigate chip multithreaded and chip multiprocessor computer architectures [1]. We concentrate on dynamic compiler optimisations to increase our architectures performance. In creating a new architecture it is possible to port an existing operating system to it. For us this was undesirable as such an operating system wouldn't expose the compiler optimisations we're concentrating on. As our optimisations were written for Java and the specific JVM of the Jikes RVM, we decided to migrate an existing Java operating system called JNODE to the Jikes RVM [13, 16].

To migrate users to a new computer architecture and operating system, the ability to execute legacy applications is desirable. Previous work has statically translated C or binary code to Java, thereby making it executable by the JVM [17, 7]. We consider this approach undesirable for the user, where a lack of intervention and the ability to get at underlying performance are key. Systems translating C to Java have to work around presenting a C programming language style memory (i.e. with pointers) in the strongly typed JVM. We have solved this problem by making our JVM also capable of executing native binary code, and working upon specific optimisation within the JVM for this purpose.

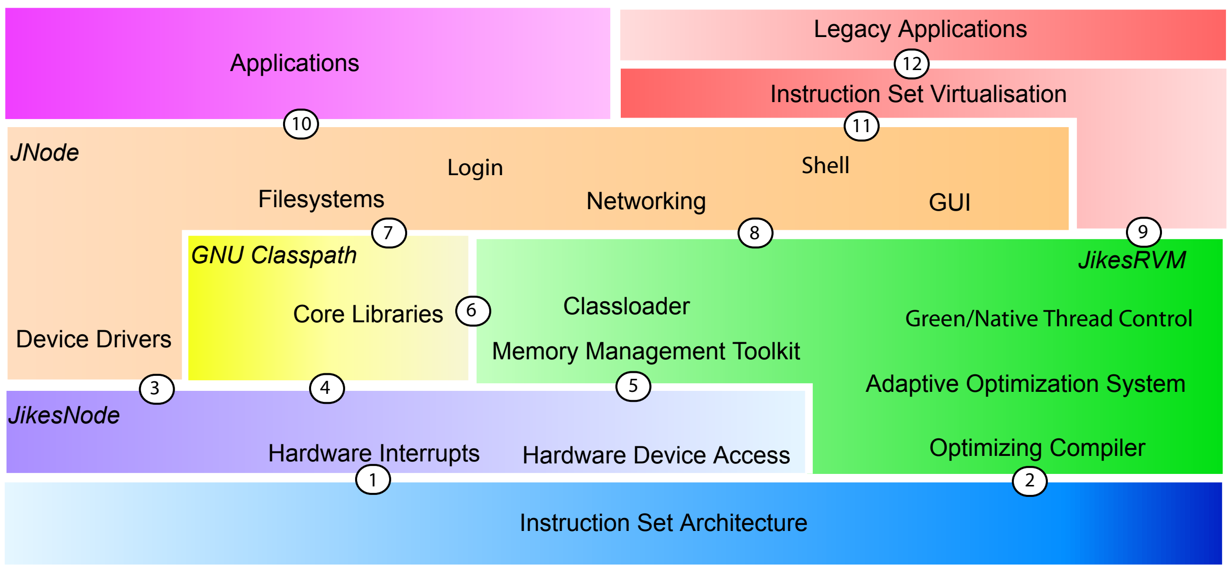

Figure 1: The JikesNODE and PearColator structure

The view of our software system is presented in figure 1. At the top of our system are applications, and at the bottom the underlying hardware and instruction set. The points at which different layers interface are labelled. GNU Classpath provides the core library and is used by the entire system, as shown at points 4, 6 and 7 [8]. To avoid possible issues with sharing the core library, JNODE presents a private copy to each application. This is achieved, in common with other multitasking JVMs [4], by binary rewriting techniques. The rewrite of the loaded class modifies the library name to also include a process identifier. Core libraries that are known to be safe are not re-written.

The JikesNODE kernel is responsible for providing the hardware interface and interrupts to the device drivers and the thread control mechanism (points 1, 3 and 5). The Jikes RVM, adaptive compilation system, is responsible for all the dynamically executing code generated by the system (point 2). The JNODE system presents the user and application interfaces on-top of the Jikes RVM system, and legacy binaries are passed to the legacy code execution environment of PearColator that directly interfaces with the Jikes RVM optimising compiler (points 8 to 12).

Legacy graphics libraries can be emulated in the system by running Java implementations of X servers [14]. Other primitive operations for sound and 3D graphics aren't yet planned to be supported.

The rest of this paper looks at how JikesNODE bootstraps the Jikes RVM as an operating system, Section 2, and how the PearColator dynamic binary translator is able to execute legacy binaries, Section 3. Finally we conclude our description of the work that is being undertaken for these two joint research efforts.

2 JikesNODE

The Jikes RVM is written in the Java programming language. To build itself it runs as a mock-up on another JVM. During this boot-image creation phase a dummy Java class is compiled. This class has sufficient dependencies for the compiler to entirely build itself. Once this compilation is complete the compiled code is written to disk into the boot-image file. A C stub program loads the boot-image file directly into the memory location it was compiled for. The C stub also provides a set of call-backs that allow access to the underlying operating system.

For JikesNODE we have created a replacement for the C stub that acts as the loader for the operating system. This stub also provides call backs that are used to diagnose the system until the Java implemented device drivers are loaded. To implement libc functions at boot time, the klibc library is used [2]. System class libraries are compiled into the boot image, but those that aren't compiled are provided in 3 jar files loaded into a RAM disk. One jar file contains the standard java class libraries, the second contains the classes of the Jikes RVM and PearColator, the final jar file contains the memory management toolkit that provides the garbage collection framework for the Jikes RVM.

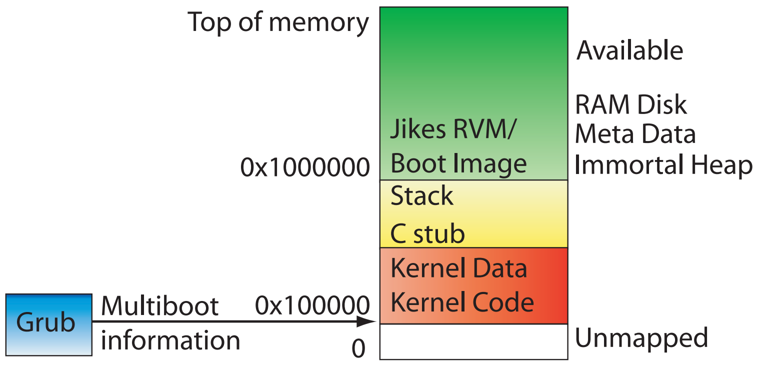

Figure 2: JikesNode memory layout

Once the system is running device drivers are loaded using the JNODE pluggable module architecture. The special compiler hooks known as virtual machine magic (VM_Magic) allow controlled access to be given by the combination of the Jikes RVM's compilers and classloader to the underlying hardware and memory system.

Our system is presently a work in progress, and currently only gets partially through its boot procedure. The efficacy of this design can be seen in existing Java operating system research such as Alta, K0, J-Kernel, Janos VM, Kaffe OS and JX [3, 9]. The multitasking virtual machine (MVM) is a multi-program and multi-user JVM using a modified HotSpot JVM [4]. By running applications as multiple users within the same virtual machine, even with extra security methods, the performance of two collaborating processes is improved by 11% for throughput and 36% for response time; the initial load time of running a second instance of an application is reduced by up to 90% [11].

We believe JikesNODE is an interesting alternative to other Java multitasking and operating system designs, thanks to the Jikes RVM. Another unique feature of the JikesNODE operating system is the integration with a legacy application execution environment. We describe PearColator in the next section.

3 PearColator

3.1 Structure

PearColator emulates the PowerPC user instruction set architecture and Linux 2.4 system call interface. PowerPC was chosen as it has a large number of registers that can be mapped to Jikes RVM HIR temporaries. This allows us to avoid creating memory operations that are expensive with our current memory system. We are working on other instruction set architectures (ISAs) which have fewer registers. We believe for these ISAs, optimisations in the Jikes RVM will be able to help to turn memory accesses to the same location into temporarily held values[6], therefore performance is hoped to be comparable across instruction sets. Linux was chosen as it was easy to discover the implementation of system calls, and the ELF executable format is relatively simple.

PearColator uses the Jikes RVM optimising compiler and adaptive framework with minimal change (roughly 30 source code lines). An initial part of the design allowed for the generated code to be turned into Java bytecodes capable of execution in any JVM. But to avoid extra translation overhead, we use the Jikes RVM directly. Figure 3 shows the interaction of the main components in the PearColator system.

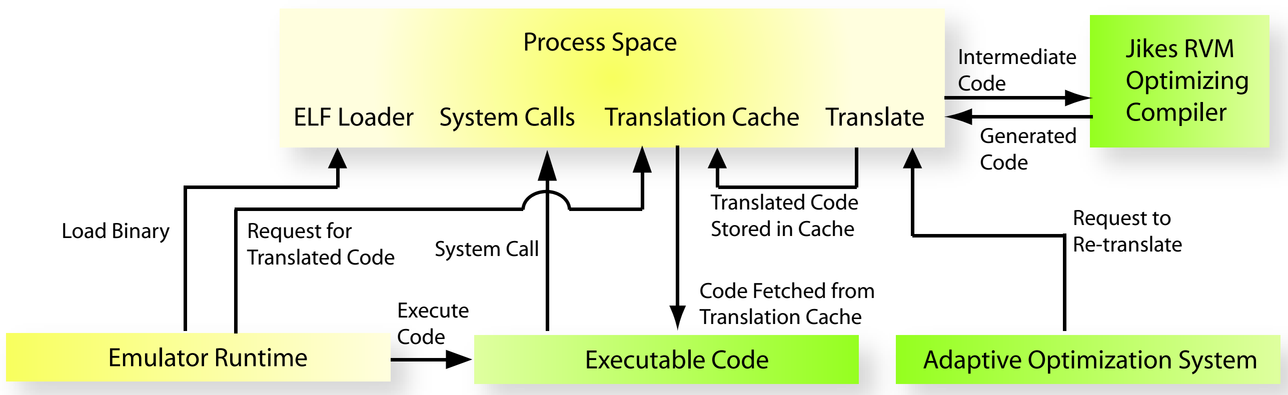

Figure 3: Overview of PearColator

3.2 Implementation

After being called upon to execute a PowerPC binary, PearColator performs an instruction to high-level IR translation. The translated instructions form a trace, as with the FX!32 and DAISY binary translators [12, 5], and maintain a set of translation assumptions for each translated instruction. These assumptions allow flag setting operations to avoid setting and then testing several bits; instead compare and branch instructions are planted. To guide trace formation, branch statistics are gathered and, if code is retranslated, used to improve the accuracy of the trace.

The memory model of the prototype PearColator uses the same address space as the Jikes RVM, and uses an emulated page table to provide the application space. The emulated page table requires the page table entry to first be calculated and loaded from memory, then the contents of that page accessed. This process adds a memory indirection to every memory access, which is undesirable. To reduce this overhead we have created an object-oriented memory implementation architecture. This architecture allows for memory to be emulated as single arrays (possibly highly optimized, for example through use of hardware segmentation) or, if the space requirements demand it, for these to be converted back into the less optimal emulated page table scheme.

The operating system interface in PearColator maps PowerPC Linux system calls into Java method calls. These method calls are planted into the translated code, and present points for method inlining. As with an application running in JikesNode, this raises the possibility of optimising a device driver into an application. However, with legacy code the memory emulation requires more analysis in the compiler than with Java's strongly typed classes.

3.3 Performance

PearColator successfully emulates a wide range of PowerPC Linux applications, with new instructions and system calls being implemented as needed. Figure 4 shows the relative performance for the Dhrystone benchmark in C and Java of PearColator, native PowerPC, PowerPC dynamic binary translators (QEMU and PearPC), JVMs and native IA32.

Profile guided optimisations allow the Java Dhrystone figures to be higher than the native C. However, the Jikes RVM's performance is not faster than C and this reflects its research nature. The performance of QEMU *fast* (no MMU emulation) shows that on a AMD Athlon XP2700+ (2163MHz) processor the performance is 30% slower than a PowerPC G3 at 600MHz. PearColator and PearPC are both slower than QEMU fast, however, they are both emulating the MMU[1].

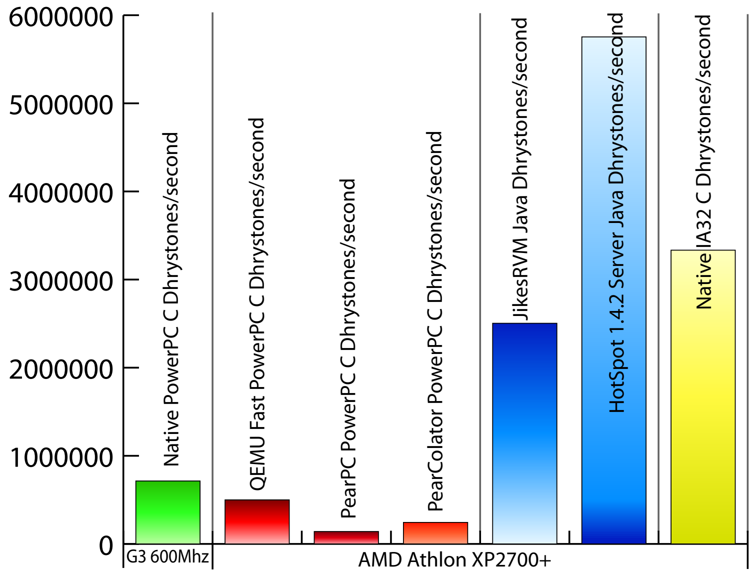

Figure 4: Overview of PearColator

Thanks to the branch predictor within the trace formation code, an entire Dhrystone loop can be moved into a trace. With an entire loop in a trace the sophistication of the Jikes RVM optimizing compiler is able to outperform the simpler compiler of PearPC by 73%. However, this result relies on careful tuning of the trace length and the fact that Dhrystone uses a lot of 32-bit values[2].

We are working on complementing the optimizing compiler approach of PearColator with an interpreter. The performance of the optimizing compiler, compared to the simpler compilers of PearPC and QEMU, we believe is

[1]PearPC's MMU emulation is accurate to that of a G3 whereas PearColator only emulates memory for a single process space.
[2]PearColator emulates byte loads and stores using 32-bit loads and stores and logical operations.

encouraging. We expect the performance of QEMU fast or better, by improving the optimisation of memory.

4 Summary and Conclusions

This paper has presented a new operating system and binary translation system based around the Jikes RVM adaptive and optimising compiler. The two systems complement each other by trying to migrate users to a higher performance, dynamically optimising, operating system architecture. Both systems are prototypes, but already performance is comparable with similar projects. Virtualisation is already an important operating system concept, we expect to see this complemented in the future by virtualisation of the instruction set, beyond what is seen in existing architectures. We expect this as the variety of things to virtualise will increase (for example, as new instruction sets are created) and the desire to further optimise hardware (possibly by simplifying).

Acknowledgements

The authors thank the developers of the Jikes RVM and JNODE. This work would not be possible without significant parts making up MSc project work. We particularly thank Georgios Gousios [10], Richard Matley [15] and Yun Zhang for their efforts. This work is funded by an EPSRC portfolio award.

References

[1] The Jamaica project. `http://www.cs.manchester.ac.uk/apt/projects/jamaica`, May 2005.

[2] Linux Kernel Archives. klibc. `http://www.kernel.org/pub/linux/libs/klibc/`, 2005.

[3] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, 2000. USENIX.

[4] Grzegorz Czajkowski and Laurent Daynes. Multitasking without comprimise: a virtual machine evolution. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 125–138, New York, NY, USA, 2001. ACM Press.

[5] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 26–37, New York, June 2–4 1997. ACM Press.

[6] Stephen J. Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 155–174, Santa Barbara, CA, 2000. Springer-Verlag.

[7] Andreas Gal, Christian Probst, and Michael Franz. Executing legacy applications on a Java operating system. In *ECOOP Workshop on Programming Languages and Operating Systems*, Oslo, Norway, June 2004.

[8] GNU. GNU Classpath. `http://www.classpath.org/`, 2005.

[9] Michael Golm, Meik Felser, and Christian Wawersich. The JX operating system. In *2002 USENIX Annual Technical Conference*, pages 45–58, Monterey, CA, June 10–15 2002.

[10] Georgios Gousios. JikesNode: A Java operating system. Master's thesis, The University of Manchester, September 2004.

[11] Janice J. Heiss. The Multi-tasking Virtual Machine: Building a highly scalable JVM. *Sun Developer Network*, March 2005.

[12] Raymond J. Hookway and Mark A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1):3–12, 1997.

[13] IBM. Jikes^{TM}Research Virtual Machine (RVM). `http://jikesrvm.sourceforge.net/`, 2005.

[14] JCraft Inc. WeirdX – Pure Java X window system server under GPL. `http://www.jcraft.com/weirdx/`, April 2004.

[15] Richard Matley. Native code execution within a JVM. Master's thesis, The University of Manchester, September 2004.

[16] Ewout Prangsma. JNODE: Java New Operating system Design Effort. `http://www.jnode.org/`, May 2005.

[17] Trent Waddington. Java backend for GCC. `http://www.itee.uq.edu.au/~cristina/uqbt.html`, 1999.