On the Future of Research VMs: A Hardware/Software Perspective

Foivos S. Zakkak, Andy Nisbet, John Mawer, Tim Hartley, Nikos Foutris, Orion Papadakis, Andreas Andronikakis, Iain Apreotesei, Christos Kotseidis
The University of Manchester
United Kingdom, M13 9PL
first.last@manchester.ac.uk

ABSTRACT
In the recent years, we have witnessed an explosion of the usages of Virtual Machines (VMs) which are currently found in desktops, smartphones, and cloud deployments. These recent developments create new research opportunities in the VM domain extending from performance to energy efficiency, and scalability studies. Research into these directions necessitates research frameworks for VMs that provide full coverage of the execution domains and hardware platforms. Unfortunately, the state of the art on Research VMs does not live up to such expectations and lacks behind industrial-strength software, making it hard for the research community to provide valuable insights.

This paper presents our work in attempting to tackle those shortcomings by introducing Beehive, our vision towards a modular and seamlessly extensible ecosystem for research on virtual machines. Beehive unifies a number of existing state-of-the-art tools and components with novel ones providing a complete platform for hardware/software co-design of Virtual Machines.

CCS CONCEPTS
• Software and its engineering → Virtual machines:

KEYWORDS
Virtual Machines, Managed Runtime Systems, Heterogeneous Systems, Simulators

ACM Reference Format:

1 INTRODUCTION
The last few years there is a movement towards providing open source modular language virtual machines. Mainly driven by the need to enable the reuse of successful components across different VMs, numerous virtual machines such as Oracle’s HotSpot [11], IBM J9 [12], .NET [19], Google v8 [8], and RPython [5], have been recently open sourced. In addition, projects like Eclipse OMR [7] and Mu [24] provide a set of core components useful to VM implementers, where each component is self-contained and able to interface with others through well-defined interfaces. This allows various combinations of different implementations of components to create a VM that matches the needs of each specific case. Graal VM [26], on the other hand, combines an efficient just in time (JIT) compiler with a language implementation framework to allow multiple languages to be efficiently implemented on top of a Java Virtual Machine (JVM).

These solutions focus on solving specific issues and often for specific environments; mainly targeting high peak performance. However, since today’s VMs are exercising a wide range of devices ranging from cell phones to powerful clusters, diverse stacks that cover all environments are needed. Such solutions need to expand beyond a single node and/or hardware architecture enabling research on complex systems (e.g. clusters or cloud) with heterogeneous machines and enormous amounts of memory [1, 10].

2 THE BEEHIVE ECOSYSTEM
Our vision is to take advantage of the movement towards open source and modular VMs and form an ecosystem of tools that will enable the next generation of research on VMs. In particular, we envision an ecosystem with the following characteristics:

• Modular and easily extensible.
• Implemented with high level languages with good IDE support and low entry barrier.
• Realistic and diverse simulation infrastructures.
• Support of multiple hardware architectures.
• Support of heterogeneous systems.
• Capability of implementing multiple languages.
• Integration with popular research tools.

We believe that such a research VM will provide a solid foundation for the research community. It will also provide a common baseline for comparing different works and it will bring the community closer, ultimately improving the quality of the conducted research.

Figure 1 visualizes the interconnection of tools that comprise the Beehive ecosystem [15]. On the top layer of the stack are the groups of applications that the ecosystem aims to improve the state of the art for. These applications range from the standard benchmark suites, to applications running on top of Big Data frameworks, including domain specific applications and implementations of managed languages.
The runtime layer (second from top) incorporates all the runtime mechanisms necessary to execute a managed language. This layer unifies, under the same compilers and runtimes, high-quality polyglot production and research VMs. It will feature two VMs, Maxine and OpenJDK HotSpot, that share a common optimizing compiler, Graal, and the Truffle runtime framework. OpenJDK HotSpot represents the production VMs, while MaxineVM [25] is a meta-circular research VM. MaxineVM will ultimately be compatible with the Java Virtual Machine Monitor Interface (JVMMC), and the JikesRVM’s Memory Management Toolkit (MMTk) [4], combining two powerful interfaces that will enable experimentation with different compilers and garbage collectors. The runtime layer will also include in-house software components and mechanisms [2, 13] that enable the acceleration of applications on specialized hardware, like SIMD hardware extensions, GPUs, FPGAs, etc. In order to support multiple ISAs, MaxineVM and its compilers are being ported to different architectures. Currently, Beehive executes on x86 and ARMv7 architectures while ongoing work extends the support for AArch64 and RISC-V via the CrossISA toolkit [14].

The layer below the runtime layer consists of the Operating System, the Services and Drivers, and the Virtualization components. The services and drivers component integrates a number of binary instrumentation tools to enable research and rapid prototyping of novel micro-architectures and ISA extensions [9, 17]. These tools will allow for code injection, binary translation and instrumentation, ISA extensions, and others.

Beehive integrates a variety of simulators providing trade-off selections between simulation speed (i.e. functional), simulation accuracy (i.e. cycle-accurate), and engineering efforts required to modify or implement new hardware timing models (i.e. software or hardware simulators). Therefore, Beehive integrates a software-based full-system, a software-based user-level, and a hardware-based user-level simulator; namely Gem5, ZSim, and APTSim respectively.

Gem5 [3] is a software-based full-system simulation framework capable of modeling numerous hardware architectures. Beehive augments Gem5 with an interface that allows it to integrate with a variety of other simulators, such as McPAT [16] (power), HotSpot [23] (thermal), Voltspot [28] (power), and NVSim [6] (circuit-level). In addition, Beehive also incorporates machine-learning and data-analytics techniques in the Gem5 simulation framework enabling detailed analysis of the output results. Finally, Beehive augments Gem5 with a fault injection framework for reliability studies that is flexible, generic/portable, supports multi-cores, and allows for reproducible experiments.

ZSim [22] combines simulation accuracy with fast simulation times on x86 (about 10 MIPS in our experiments) and allows the execution of arbitrary managed workloads via lightweight user-level virtualization. MaxSim [20] is a software-based simulation platform for x86-64 built on the Beehive ecosystem using the ZSim simulator. MaxSim can perform fast and accurate simulation of managed runtime workloads running on top of the Maxine VM and its capabilities include: 1) low-intrusive microarchitectural profiling via pointer tagging on x86-64 platforms, 2) modeling of hardware extensions related, but not limited to, tagged pointers, and 3) modeling of complex software changes via address-space unfolding. Rodchenko et al. [21] use MaxSim to demonstrate that hardware software co-designed pointer tagging based optimizations can be used to eliminate the type information pointer from an object’s storage.

APTSim [18] is a hardware-based simulator for ARM2 architectures, implemented on Xilinx Zynq platforms [27]. Beehive through APTSim can directly interface unmodified application executables with FPGA hardware intellectual property (IP) and thus accelerate simulation. In APTSim functional simulation occurs natively on the ARM cores, and cycle-based timing is performed using FPGA hardware models of the memory system hierarchy and the CPU micro-architecture. The functional simulation, or an external tool must produce a trace of address loads/stores, and any program counter (PC) changes that are consumed by the FPGA timing models. In Beehive we achieve this through MaxineVM’s optimizing compiler that instruments the application during JIT compilation.

Combining the above, the Beehive ecosystem will allow for full system co-design and exploration leading research towards improved performance, energy efficiency, and resiliency. The Beehive ecosystem is an ongoing project and its components are gradually being open-sourced at https://github.com/beehive-lab. Each individual component is being validated against standard methodologies and provide full coverage of the community-accepted benchmarks.

**ACKNOWLEDGMENTS**

This work is partially supported by the EPSRC grant PAMELA EP/K008730/1, and the EU Horizon 2020 grant ACTiCLOUD 732366.

---

1https://github.com/beehive-lab/MaxSim
2It is equally applicable to any valid event trace and CPU ISA, as long as CPU microarchitecture models are available.