ACTiCLOUD: ACTivating resource efficiency and large databases in the CLOUD

Project No: 732366

H2020-ICT-2016-1

Technical Report on Hyperscale JVM v1.0

18/12/2018

Executive summary:

This technical report is based on ACTiCLOUD's Deliverable 3.2 that provides the initial version of ACTiCLOUD's custom Java Virtual Machine implementation, named as Hyperscale JVM. The technical report describes the software source code and its documentation.

List of authors:

Author	Affiliation
Christos Kotselidis	UNIMAN
Foivos Zakkak	UNIMAN

ACTiCLOUD Consortium:

Participant No	Participant organisation name	Short name	Country
1 (Coordinator)	Institute of Communication and Computer Systems	ICCS	Greece
2	Numascale AS	NSCALE	Norway
3	Kaleao Limited	KALEAO	UK
4	OnApp Limited	ONAPP	Gibraltar
5	University of Manchester	UNIMAN	UK
6	MonetDB Solutions B.V.	MDBS	Netherlands
7	Neo Technology	NEO	Sweden
8	UMEA University	UMU	Sweden

Table of Contents

Figures

Tables

List of Abbreviations

Abbreviation	**Meaning**
CI	Continuous Integration
DOA	Description of Action
QA	Quality assurance
WP	Work Package
WPL	Work Package Leader
GC	Garbage Collection
ISA	Instruction Set Architecture
JVM	Java Virtual Machine
HJVM	Hyperscale JVM
LOC	Lines of Code
SA	Software Artifact

1 Introduction

ACTiCLOUD's vision is to develop a novel cloud architecture that will break the existing scale-up and share-nothing barriers and enable the holistic management of physical resources both at the local cloud site and the distributed levels, targeting drastically improved utilization and scalability of resources. This will ultimately translate to: a) significant cost and performance improvements for CSPs, b) higher performance stability and lower pricing for cloud applications, and c) enhanced flexibility and scalability of cloud resources for intensive database applications that have until now faced tough challenges in covering their resource demands from existing cloud offerings.

ACTiCLOUD brings together prestigious academic institutions with extensive expertise in addressing R&D challenges in the areas of large-scale cloud architectures, distributed systems and software, with industrial partners whose products span the entire stack of cloud computing with technologies that break through today's scale-up and share-nothing limitations, with server architectures, cloud system software and up to cutting-edge database systems. By joining these forces, we aim to enhance the viability of cloud deployment scenarios through enhancement of the various technology ingredients, i.e. the hypervisor, the cloud manager, system libraries, language runtimes and database systems with a novel and holistic set of mechanisms and policies built on top of these new-generation computing system architectures and therefore enabling a distributed, hyper-converged, "share-anything", resource scale-out cloud platforms to broaden the applicability of cloud technology across more markets through richer and more cost effective application deployments.

In ACTiCLOUD, the University of Manchester undertakes the role of designing, implementing and optimizing Java Virtual Machines (JVMs) for the server architectures proposed by ACTiCLOUD. In this technical report, the current progress of the proposed HyperScale JVM is presented along with the technical outcomes during the first implementation period of ACTiCLOUD with respect to Task T3.2 of WP3.

2 Overview

In this section we briefly revise the task and the deliverable descriptions and we describe how the presented work relates to the project's Strategic Objectives and use cases of the partners.

2.1 Task 3.2: Hyperscale managed runtimes

Task 3.2 and in general the undertaken work in the context of HyperScale JVMs is based on OpenJDK's compilers (Graal). Since Graal is shared among both industrial and research VMs, it allows us to perform versatile research and development on different contexts. While, a part of the work will be conducted on state-of-the-art industrial-strength JVMs (OpenJDK), significant effort is also placed on bringing up a state-of-the-art research VM (MaxineVM) transitioning it to ACTiCLOUD's envisioned Hyperscale JVM (HJVM).

Sections 2.2 and 3 describe in detail the necessary steps to achieve HJVM as well as our progress until M18. The resulting HJVM will enable us to not only perform detailed and accurate research on the underlying novel architectures but also increase the impact on the research community as it will be the first JVM capable of such functionalities.

2.2 In this Technical Report

As already mentioned, the dual approach that we follow on ACTiCLOUD regarding the research and development of HJVMs have generated several software artifacts that span across different JVM implementations. Table 1 lists those along with their relation to both the Strategic Objectives (SO) and the use cases of the partners.

Strategic Objective 1 (SO1): Effective utilization of cloud resources.

Strategic Objective 2 (SO2): Deployment of resource demanding applications in the cloud.

Table 1: List of D3.2 software deliverables and relation with SOs and use cases.

No	Software Artifact	Description	Relation to SO	Relation to use cases
1	Transition of MaxineVM to Java8	During this work we developed the necessary software changes for transitioning MaxineVM to Java8 (previously on Java7).	SO1	Necessary to execute Neo4J on the HJVM.
2	Continuous Integration (CI) Framework	Implemented a CI integration framework for stability and regression testing during ACTiCLOUD.	Indirectly to SO 1, 2	Stability and performance testing of new optimizations for all use

				cases.
3	AArch64 port of MaxineVM	Ported the MaxineVM to the ARM AArch64 architecture.	SO 1	Necessary to execute all use cases on the Kaleao KMAX platform.
4	Integration with MMTk	Integration of MaxineVM with the Memory Management Toolkit (MMTk) for fast prototyping and experimentation of Garbage Collection Algorithms.	SO 2	Necessary to implement NUMA-aware GC optimizations on both Numascale and KMAX architectures.
5	GarbageBench	Implementation of a synthetic Garbage Collection benchmark for feasibility studies and experimentation.	SO 1, 2	Necessary to isolate GC performance metrics and assess the memory optimizations of HJVM for all use cases and platforms.

As shown in Table 1, numerous developments have taken place in different fronts and contexts with the ultimate goal being the release of the HyperScale JVM in the final phase of the project.

Figure 1 illustrates how the above software artifacts compose the resulting HJVM (described in more detail in Section 3).

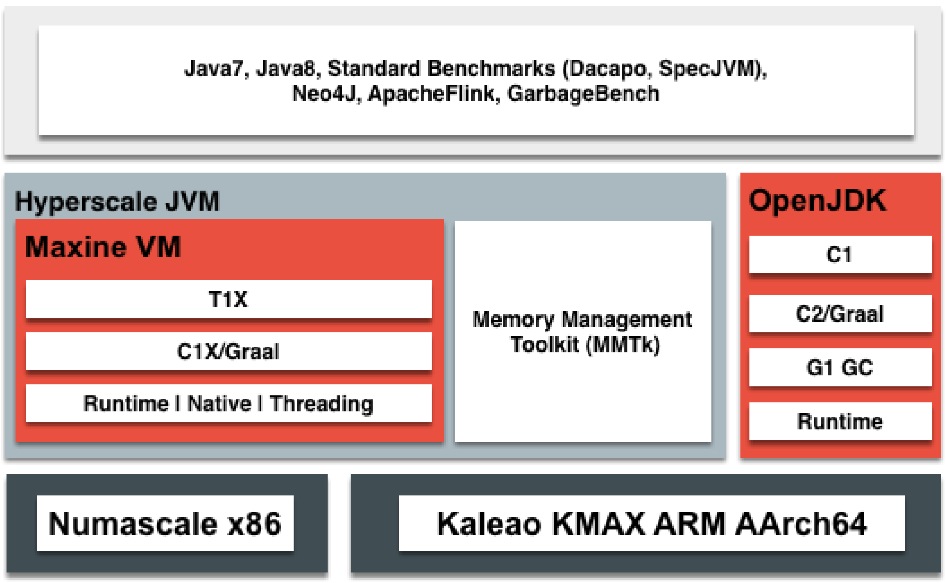

Java7, Java8, Standard Benchmarks (Dacapo, SpecJVM), Neo4J, ApacheFlink, GarbageBench
Hyperscale JVM
Maxine VM
T1X
C1X/Graal
Runtime I Native I Threading
Memory Management Toolkit (MMTk)
OpenJDK
C1
C2/Graal
G1 GC
Runtime
Numascale x86
Kaleao KMAX ARM AArch64

Figure 1: Relation between Software Artifacts (SAs) of Table 1 and HJVM

As shown in Figure 1, HJVM consists of the integration of MaxineVM with MMTk along with its execution capabilities on the Numascale and KMAX platforms. HJVM, relates with OpenJDK by employing the same optimizing compiler (Graal). In addition, HJVM can execute Java7 and Java8 workloads including standard benchmarks, Neo4J, Apache Flink, and GarbageBench (SA 5).

2.3 Relation to ACTiCLOUD's Objectives, Use Cases and Business Scenarios

As shown in Table 1, all SAs contribute directly or indirectly to ACTiCLOUD's Strategic Objectives (SOs). All SAs that ultimately constitute parts of the HJVM enable not only the execution of the partners' use cases on the ACTiCLOUD architecture but will also provide a platform for experimentation and performance optimizations. Table 1 explains how each of the software components enables ACTiCLOUDs use cases. Regarding the SOs, the HJVM integrated with the rest of the layers of ACTiCLOUD contributes to both SOs as it will enable both increased performance and efficient resource utilization.

3 The Hyperscale JVM

3.1 Generic Description

Figure 1 depicts the Hyperscale JVM (HJVM) and its surrounding components as envisioned by ACTiCLOUD. As shown, it consists of two main components: MaxineVM and MMTk. In addition, the JVM has to run on both x86 and AArch64 architectures in order to utilize the hardware platforms of the ACTiCLOUD architecture (Numascale-x86 and Kaleao KMAX-AArch64). Finally, the HJVM will have to be able to execute the workloads of interest to ACTiCLOUD; namely Neo4J, Apache Flink and standard benchmarks. In addition to the above the HJVM must be able to run on the hypervisors provided by OnAPP.

Initial Status

To reach a fully functional state of the HJVM shown in Figure 2, the SAs (steps) of Table 1 had to be completed. The starting point of our work was MaxineVM running on Java 7 on the x86 architecture. Evidently, a significant development effort had to be made to transition to an operational stage of HJVM before we explore any optimizations. The next subsection (3.2) details those improvements.

3.2 Improvements and Updates within ACTiCLOUD

Java 8 support

Java 8 support is critical for the HJVM since it is essential to run Neo4J workloads. Neo4J relies heavily on Java 8 and therefore it was imperative that the HJVM would also support it.

Transitioning HJVM from Java 7 to Java 8 was a complicated process since this particular update revision of the JVM was a major one (compared to previous releases). The key feature of Java 8 is the support of lambda expressions which is supported by a new Java bytecode (namely *invoke_dynamic*) that enables the execution of method handles. The implementation of these features propagates throughout the JVM since all compilers, runtime layers, and JDK support must be updated to support the new bytecodes and work to completion to support Java 8.

The low-level nature of this work[1], since all compilers have to be augmented as well as inline assembly has to be added, created a number of implementation challenges during its development. Finally, we successfully managed to transition HJVM to Java 8 as of version 2.2 (https://github.com/beehive-lab/Maxine-VM/releases/tag/v2.2.0) with the HJVM now successfully building with Java 8. Please note that although we maintained backwards compatibility with Java 7, a decision has been made to drop support for Java 6 (and backwards) due to implementation and support complexity. The current version of the HJVM supports both Linux and OSX platforms on Java 8 and the pass-rates as well as performance results on standard benchmarks (Dacapo-9.12 and SpecJVM2008) can be found here: https://maxine-vm.readthedocs.io/en/latest/Status.html.s

The work carried out during the ACTiCLOUD project regarding the transitioning of MaxineVM to Java 8 resulted in MaxineVM being the only research VM available that is able to execute advanced features of the Java programming language. In addition, this transition creates new research opportunities for enabling HJVM polyglot JVMs (VMs that support more than one

[1] e.g. https://github.com/beehive-lab/Maxine-VM/commit/c7e0b198a3785f9fdca9b0c02c32c7aaae9958d8

programming language) as described in our paper published this year at the MoreVMs workshop where we set ACTiCLOUD's vision regarding future research VMs[2].

Continuous Integration (CI) Infrastructure

Although the work described in this subsection is not reflected in the DOA of ACTiCLOUD, we include it for completeness and in order to highlight the impact that SA 1 had in the VM research community.

Upon successfully transitioning to Java 8, MaxineVM became the only research VM capable of supporting modern workloads and applications such as Neo4J. The proper dissemination of our results, through the ACTiCLOUD's channels, resulted in an increased interest of external VM developers and researchers.

Until that point in time, the majority of researchers have been using JikesRVM[3] as their platform of choice. However, since JikesRVM does not support Java 8, numerous developers have shifted their interest to MaxineVM. Combining this development with our ongoing work on MMTk integration (originated in the context of JikesRVM) as well as the plethora of platforms that MaxineVM supports (SA3), CI tools have to be developed in order to ease our workflows, verification and validation processes, and regression testing.

Therefore, proper CI tools based on the Jenkins framework[4], have been developed and integrated with Github and ACTiCLOUD's Slack channels in order to support external collaborations as HJVM is gaining attention in the research community. In addition, the implementation of the CI framework will help us in continuing supporting HJVM after ACTiCLOUD's completion as stated in the DOA.

AArch64 support

The support of the AArch64 ISA (and architecture) is one of the key enablers of the HJVM since its vital in order to utilize the KMAX architecture. Porting MaxineVM to the AArch64 architecture entailed a significant development and debugging effort that concentrated in successfully porting both of its compilers (namely T1X and C1X), the runtime and native layers, as well as implementing all compiler stubs and adapters that are mainly written as inline assembly.

During the activities of porting MaxineVM to the AArch64 architecture, many challenges have been encountered and a number of tools and frameworks have been developed in order to aid the effort. During this activity we researched and developed the CrossISA Tookit[5]: a software framework to assist in the development and debugging efforts of Virtual Machines. The CrossISA Toolkit has been developed following a modular approach which makes possible for it to attach

[2] Foivos S. Zakkak, Andy Nisbet, John Mawer, Tim Hartley, Nikos Foutris, Orion Papadakis, Andreas Andronikakis, Iain Apreotesei, Christos Kotselidis. ***On the Future of Research VMs: A Hardware/Software Perspective***. In MoreVMs 2018 Workshop on Modern Language Runtimes, Ecosystems, and VMs, April 2018.

[3] https://www.jikesrvm.org

[4] https://jenkins.io

[5] Christos Kotselidis, Andy Nisbet, Foivos S. Zakkak, and Nikos Foutris. ***Cross-ISA debugging in meta-circular VMs***. In the 9th International Workshop on Virtual Machines and Intermediate Languages (VMIL '17), October 2017.

to any Virtual Machine in a "plug-and-play" fashion. The following paragraph describes in detail the developed framework as well as the challenges it tackles.

CrossISA Toolkit

Extending current Virtual Machine implementations to new Instruction Set Architectures (ISAs) entails a significant programming and debugging effort. Meta-circular VMs add another level of complexity towards this aim since they have to compile themselves with the same compiler that is being extended. Therefore, having low-level debugging tools is of vital importance in decreasing development time and bugs introduced.

The problem of meta-circularity

The fact that Maxine VM's bootimage is ahead-of-time compiled by one of its runtime compilers, creates a paradox when trying to port it to a different ISA. This paradox along with the fact that Maxine VM is implemented in the same programming language it executes -Java- results in its meta-circular nature. Ideally, to test the validity of the generated code for the new ISA, we would like to run the VM and through its unit testing framework start testing individual compiled methods under the new ISA. However, this is not possible in our case since we cannot boot the VM if we do not ensure that the compilers have been ported to the new ISA. At the same time, we cannot test the compilers through the VM's testing framework if we cannot boot the VM.

MaxineVM (and consequently HJVM), being itself a metacircular VM, falls into this category and therefore a set of tools and methodologies had to be developed to assist the porting activities to AArch64.
During that process, we developed a QEMU-based toolchain which enables us to debug a wide range of VM features in an automated way. The developed and published CrossISA toolchain[6] has been integrated with the JUNIT-based testing framework of Maxine VM and can execute from simple assembly instructions to fully JIT compiled code. Furthermore, it is fully open-sourced and can be adapted to any other VMs seamlessly[7]. Finally, a compiler-assisted methodology that helps us identify, at runtime, faulty methods that generate no stack traces, in an automatic and fast manner has been developed.

Figure 2 outlines the developed CrossISA toolkit. As shown, the existing unit testing framework of Maxine VM communicates with the MaxineTester class which is responsible for the whole coordination. In order to run a unit test the following steps are performed in order:

1. MaxineTester Initialization: When the unit tests are invoked, they initialize (1) the MaxineTester which resets all internal state and QEMU output files. This process is performed per unit test by invoking the initialize call before every unit test (using the

[6] Christos Kotselidis, Andy Nisbet, Foivos S. Zakkak, and Nikos Foutris. ***Cross-ISA debugging in meta-circular VMs***. In the 9th International Workshop on Virtual Machines and Intermediate Languages (VMIL '17), October 2017.

[7] https://github.com/beehive-lab/Maxine-VM

JUNIT's @Before annotation). This is due to instabilities we observed during QEMU execution which led to the unit testing framework hanging during nightly regressions. Hence, we followed a more conservative approach in which the QEMU is re-initialized during every unit test execution. After the initialization completes a code buffer is returned to the unit test (2) which serves as the placeholder for the generated assembly code.

2. Generating the code of the unit test: Depending on the nature of the unit test; i.e., simple assembly instruction, T1X or C1X testing, the code buffer is filled in a different manner. When the code buffer is filled, it is passed to the MaxineTester for QEMU emulation (3).
3. Composing the binary for QEMU emulation: After receiving the code to be tested, the MaxineTester assembles the binary that will be passed to QEMU for emulation (4). The process of creating the binary file (test.bin) is as follows: Initially, we generate the assembly code of two helping assembly files (asm_startup.s and entry.s) which are linked together with the binary code of the unit test. Consequently, the code buffer that contains the assembly code of the unit test is inlined to a C file (codebuffer.c) and a function pointer to its first entry is installed. Finally, the actual test (test.c) that links together the codebuffer.c and the two assembly files (asm_startup.s and entry.s) is compiled and the test.bin binary is formed. The generated test.bin binary, in essence, contains code that helps running the binary code injected through the code buffer (i.e. a main function with a function pointer invocation to the embedded code of the code buffer).
4. Performing the QEMU emulation: The next step, after creating the test.bin binary file that contains our unit test, is the QEMU emulation (5). In our case, since we port Maxine VM to the AArch64 ISA, we simulate a Cortex-A53 processor. As shown in Figure 2, QEMU will run the binary emulating the ARM core we defined, and upon completion, it will dump the register file to an output file defined in the MaxineTester class.
5. Validating the execution output: The output of QEMU executing a unit test is the dump of the register file of the emulated processor. The output register file is validated against the expected values as set in the unit test definition. Depending on the nature of the unit test, such definitions might be explicit (e.g. in which register we expect a certain value to be written) or implicit (e.g. the return value of a C1X compiled method which is written in register r0 according to the ARM calling convention).

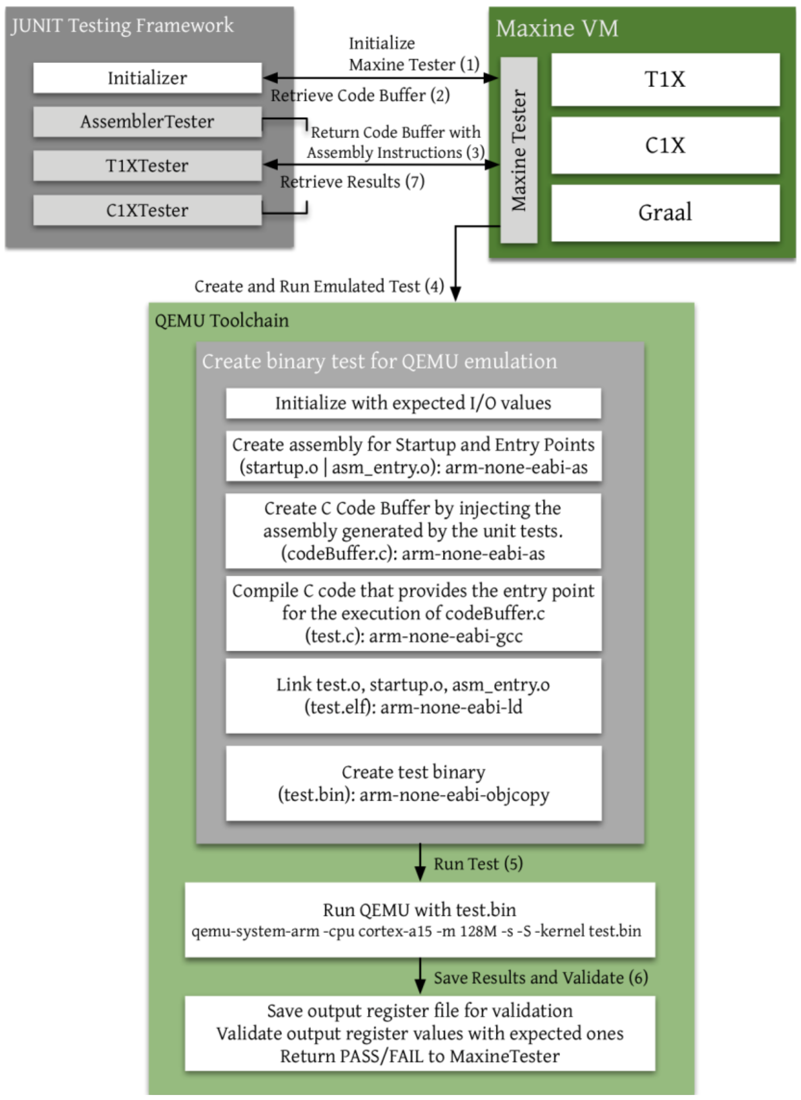

JUNIT Testing Framework
Initializer
AssemblerTester
T1XTester
C1XTester
Initialize Maxine Tester (1)
Retrieve Code Buffer (2)
Return Code Buffer with Assembly Instructions (3)
Retrieve Results (7)
Maxine VM
Maxine Tester
T1X
C1X
Graal
Create and Run Emulated Test (4)
QEMU Toolchain
Create binary test for QEMU emulation
Initialize with expected I/O values
Create assembly for Startup and Entry Points (startup.o | asm_entry.o): arm-none-eabi-as
Create C Code Buffer by injecting the assembly generated by the unit tests. (codeBuffer.c): arm-none-eabi-as
Compile C code that provides the entry point for the execution of codeBuffer.c (test.c): arm-none-eabi-gcc
Link test.o, startup.o, asm_entry.o (test.elf): arm-none-eabi-ld
Create test binary (test.bin): arm-none-eabi-objcopy
Run Test (5)
Run QEMU with test.bin
qemu-system-arm -cpu cortex-a15 -m 128M -s -S -kernel test.bin
Save Results and Validate (6)
Save output register file for validation
Validate output register values with expected ones
Return PASS/FAIL to MaxineTester

Figure 2 : The CrossISA toolchain.

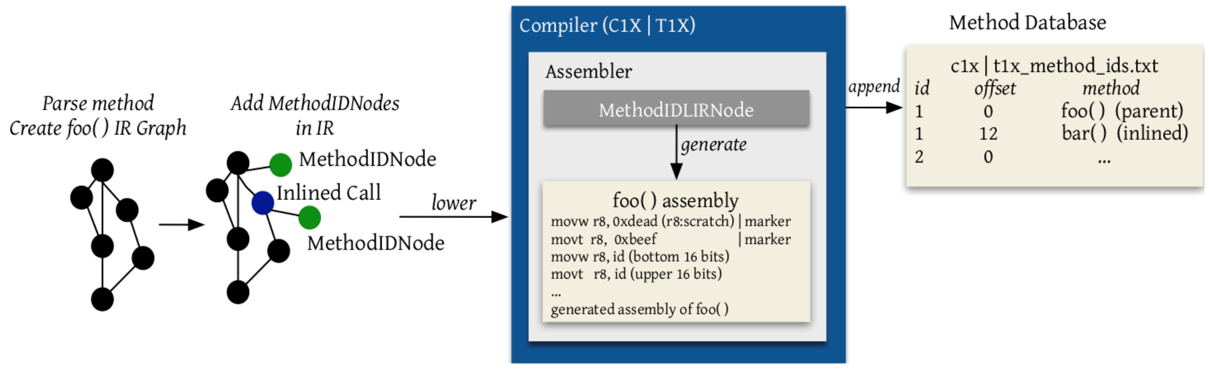


Figure 3 : Adding MethodIDs to IR graph and generation of method database during compilation.

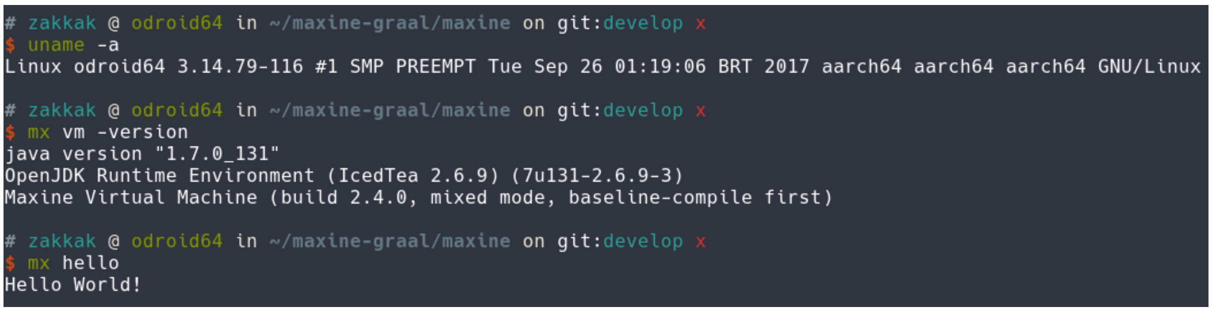
When boostraping a meta-circular VM on a new ISA, many parts of the VM are still untested. Bootstraping the Maxine VM (HJVM) is a multi-stage process in which various parts of the VM are sequentially initialized. Since the exception handling mechanism of Maxine VM is initialized at later stages of the boot process, we encountered numerous hard segfaults that generated no exception stack traces. In order to identify such faulty methods during runtime, we revised the compiler-assisted methodology shown in Figure 3.

Since most of the times the VM was failing with a hard SIGSEGV violation, we were running it inside gdb in order to read the program counter (PC) and retrieve the address of the faulty instruction. Our next objective was to be able to map that address to the faulty compiled Java method in an automated manner. For that reason, we augmented Maxine VM's compilers (both T1X and C1X) to inject, during compilation, a unique identifier called MethodID.

The MethodIDs always start from a predefined number and are incremented in a thread-safe manner, using compare-and-swap instructions, during a method's compilation. In the generated code, the MethodID manifests as a pair of mov instructions to the scratch register following another pair of mov instructions that serve as a signature that we can later search for. At the same time, the compiler records that unique MethodID along with the method signature in a text file, essentially creating a database that maps MethodIDs to Java method signatures.

Upon a crash, we display the instruction sequence backwards inside gdb and search for the MethodID signature. Then, we inquire the MethodID inside the generated text file and detect the name of the faulty method.

With the help of the CrossISA toolkit we managed to successfully port the entire VM to the AArch64 architecture being able to execute since version 2.4[8], HelloWorld on an AAArch64 odroid-c2 board, shown in Figure 4.


```
# zakkak @ odroid64 in ~/maxine-graal/maxine on git:develop x
$ uname -a
Linux odroid64 3.14.79-116 #1 SMP PREEMPT Tue Sep 26 01:19:06 BRT 2017 aarch64 aarch64 aarch64 GNU/Linux

# zakkak @ odroid64 in ~/maxine-graal/maxine on git:develop x
$ mx vm -version
java version "1.7.0_131"
OpenJDK Runtime Environment (IcedTea 2.6.9) (7u131-2.6.9-3)
Maxine Virtual Machine (build 2.4.0, mixed mode, baseline-compile first)

# zakkak @ odroid64 in ~/maxine-graal/maxine on git:develop x
$ mx hello
Hello World!
```

Figure 4 : MaxineVM (HJVM) successfully booting and executing HelloWorld on an AArch64 core.

The successful bootstrapping and execution of the HelloWorld example of MaxineVM on AArch64, is a significant milestone towards increased pass-rates and complex workload execution. This is due to the booting of MaxineVM being a complex process as Maxine itself is a Java program exercising advanced execution paths until it dynamically loads a class for application execution. Such paths include the employment of both T1X and C1X compiled code, the execution of the native substrate, and numerous compiler stubs and adapters all programmed in native AArch64 assembly.

The next step after successfully booting MaxineVM on AArch64 is to achieve >95% pass-rate of the JTT benchmarks (footnote) and >95% on Dacapo and SpecJVM. In addition, Neo4J and Apache Flink will also be successfully executed on MaxineVM on AArch64 on the Kaleao KMAX platform. We anticipate that the above will be completed by M21 of ACTiCLOUD.

[8] https://github.com/beehive-lab/Maxine-VM/releases/tag/v2.4.0

Currently, we achieve the following pass rates:

- JTT Tests (~3000): ~99.7%
- SPECjvm2008 (single threaded): ~84%
- DaCapo (single threaded): ~61%

MMTk Integration

The integration of MaxineVM with MMTk is the final and most important step towards the transition to the HJVM. To help the reader understand the reasoning behind this design decision and its importance, we provide some background information regarding MMTk and its significance in the VM research community.

The Memory Management Toolkit (MMTk)

MMTk[9] is a collection of GC algorithms developed initially in the context of JikesRVM. MMTk offers not only a set of high performing GC algorithms but it also provides the basic building blocks for implementing new ones. MMTk currently is integrated with JikesRVM and for that reason the latter attracts the vast majority of the researchers in the automatic memory management research field. In the context of HJVM, we were faced with the following dilemma: either implement our own toolkit equivalent to MMTk, or port MMTk on MaxineVM. We opted for the second option due to limited implementation time and the potential impact of the final integration leading to HJVM. As illustrated in Figure 1, HJVM will ultimately exercise the state-of-the-art from both the VM (Maxine VM) and the GC (MMTk) sides. Therefore, we anticipate that the generated impact will result in shifting the majority of the researchers to HJVM making it the first choice of the research community. Most importantly, HJVM will leverage the advanced GC functionality and versatility of MMTk allowing us to experiment with numerous GC algorithms and configurations on the Numascale and Kaleao platforms in the remaining duration of the project.

Integrating MaxineVM with MMTk is a challenging process due to the large code-bases of both projects; collectively over 600K lines of code. In addition, the integration of these two frameworks occurs in various places in their source code and therefore careful design and consideration needed to be made before the integration process started.

After we studied the two code-bases we identified the following connection points and integration steps (tasks) that were/are necessitated:

1. **Integration of MMTk in MaxineVM's build process:** This task includes the dissection of the MMTk source code from JikesRVM's code-base, its packaging and integration with MaxineVM's build toolchain. **Completed**
2. **Unification of addressing types:** Both MMTk and MaxineVM use internally a set of classes that represent address types. Notions such as Address, Offset, Word, etc., are represented internally by classes and objects that are handled by the compilers in a special manner. To avoid memory indirections when loading addresses, the compilers replace those objects references with casted versions of their contents and they are ultimately translated as memory references in the compiled code. In order to unify,

[9] https://github.com/JikesRVM/JikesRVM/tree/master/MMTk

MMTk and MaxineVM we had to translate MMTk's representations of addresses to MaxineVM ones. **Completed**

3. **Integration of MMTk's memory allocation paths into Maxine VM's compilers (T1X and C1X):** This task entails the integration of the memory allocation paths of MMTk to MaxineVMs. By proper code inlining we will manage to fold away all indirect calls and create a dense instruction sequence of small length utilizing MMTk's allocation paths in an efficient manner. **Partially complete.**
4. **Thread synchronization between application (mutator) and GC threads:** This task entails the implementation of the thread orchestration and synchronization policies between application (mutator) threads, that reside on MaxineVM, and GC threads that reside in MMTk. Proper safepoint injection and synchronization is essential to avoid memory leaks and inconsistent states. **Not started yet.**
5. **Integration of MMTK's metadata and functionality (e.g. write barriers, card tables, remember sets, etc.) into MaxineVM:** This is the last segment of the integration that require all previous four steps to be completed. **Not started yet.**

Currently we have successfully completed Steps 1-2 with 3-5 remaining until completion. The integration code is not yet published on MaxineVM's public master branch.

GarbageBench

In parallel with the activities related to HJVM, since the very early stages of the project we started performing performance analysis of ACTiCLOUD's workloads to identify if a number of optimizations could be performed quickly. The objective was to propose a set of optimizations that could be integrated both to OpenJDK and HJVM focusing on Neo4J and Apache Flink on both Numascale and KMAX architectures.

We started our evaluation on Numascale exercising initially standard benchmarks (Dacapo and SpecJVM) typically used in JVM research. However, we soon realized that these benchmarks cannot fully take advantage of such large aggregated architectures since they cannot utilize the available TBs of memory. Consequently, we started experimented directly with Neo4J workloads (as well as Apache Flink) where we could arbitrarily scale the JVM in great memory lengths.

Both Neo4J and Apache Flink being Java applications themselves, are executed on top of JVMs. Therefore, their performance analysis follows standard Java performance methodologies techniques that alone constitute a separate branch of JVM research.

Performance Analysis and Methodologies of Managed Runtime Systems

Managed Runtime Systems, such as the JVM, require a very different approach when assessing their performance compared to static languages such as C or C++. End-to-end execution times do not regard only application code but entail various overheads imposed by the runtime such a GC, compilation, finalization and others. Therefore, when assessing performance of applications running on the JVM, a clear distinction must be made about the metrics and the stage of execution the numbers concern. Typically, a "warmup" period where "hot" code is being "Just-In-Time (JIT)" compiled from their interpreted equivalents, is followed by "peak" performance periods where performance stabilizes as most of the code has been compiled. Nevertheless, runtime-related overheads such as GC and or de-optimizations are inevitably measured as part of the end-to-end execution time.

When we started analyzing the performance of Neo4J, we soon realized that it was extremely hard to isolate GC and application times to propose optimizations that regards one or the other. Especially, when exercising TBs of main memory, as in our case with Numascale, extreme levels of noise are factored into the results that makes the research and development of optimizations almost impossible.

To that end, we took the decision to develop GarbageBench: a state-of-the-art GC benchmark that can synthetically simulate different workloads, data structures and memory access partners in an isolated way.

GarbageBench, currently at a pre-alpha release state, aims to fill the gap in VM research with respect to GC benchmarking. Both its design and implementation allow the on-demand creation of synthetic workloads that can emulate access partners and behaviors found in real-world application. In contrast to the real-world applications, however, GarbageBench can stress only the memory allocation and GC subsystems of the VM, enabling proper and isolated evaluation of the proposed optimizations factoring out the rest of the VM and application subsystems that may influence the results.

GarbageBench currently supports the following key features:

- Custom heap utilization: Users can define how large the workload should be in terms of utilized memory.
- NUMA-aware thread and memory placement: Users can select among thread placement techniques and memory allocation placements. This is very important especially in the context of ACTiCLOUD where NUMA-aware thread and memory placement are key enablers of the HJVM.
- Adjustable load/store operations: Users can select the ratio between load/store operations on the selected data structure.
- Numerous data structures: Users can define a particular data structure or a set of data structures to be used during execution. This way, we can simulate different workloads and access partners. For example, Neo4J which is a graph database can be simulated by using the graph data structure of GarbageBench.

Figure 5 shows the UML diagram of the current design of GarbageBench. Our first priority is to validate GarbageBench against the memory behavior of Neo4J and open source the suite. Naturally, more additions and enhancements will be made during the duration of ACTiCLOUD with a clear objective being to establish GarbageBench as the benchmark of choice for JVM research.

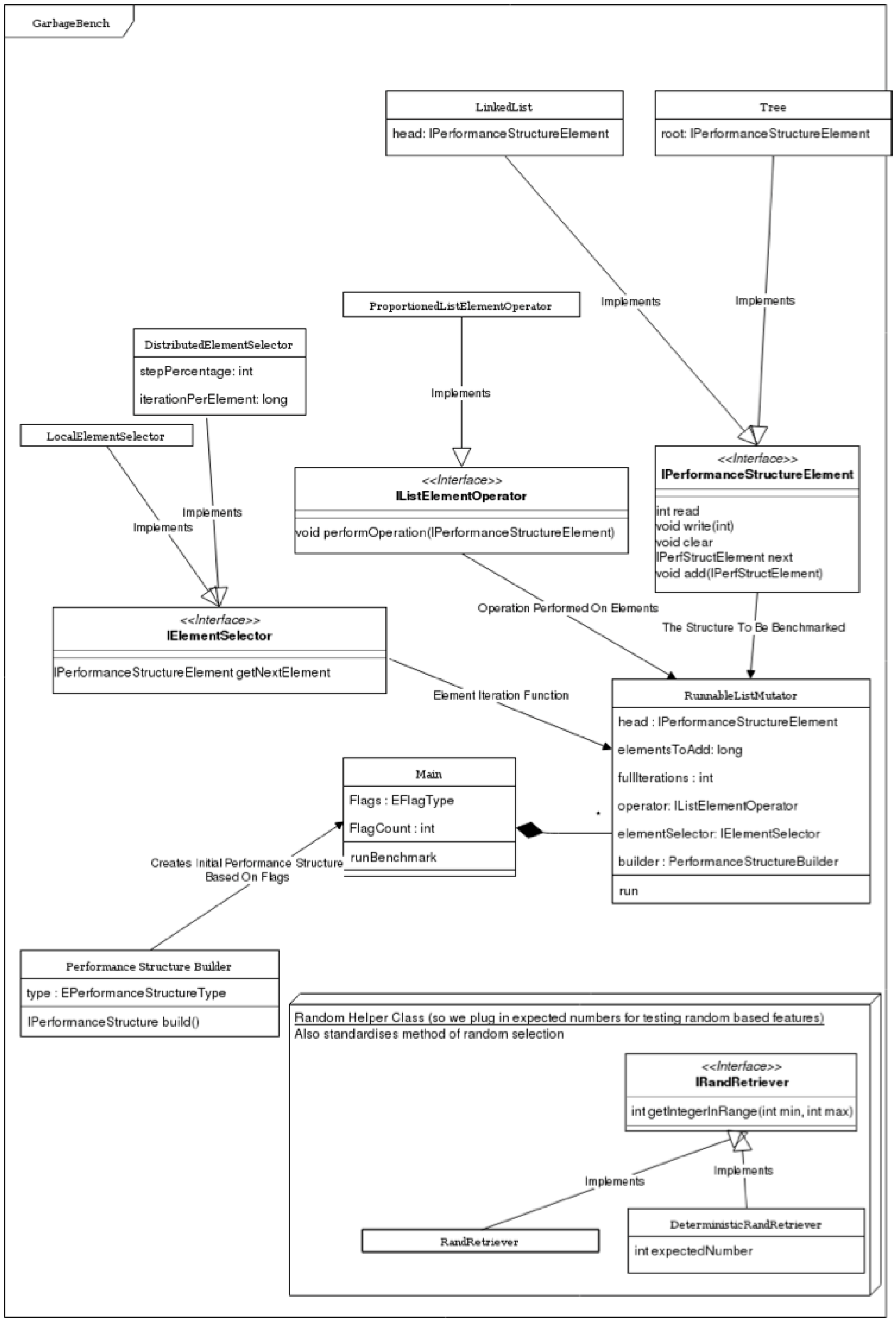

GarbageBench
LinkedList
head: IPerformanceStructureElement
Tree
root: IPerformanceStructureElement
Implements
Implements
ProportionedListElementOperator
Implements
DistributedElementSelector
stepPercentage: int
iterationPerElement: long
LocalElementSelector
Implements
Implements
<<Interface>>
IListElementOperator
void performOperation(IPerformanceStructureElement)
<<Interface>>
IPerformanceStructureElement
int read
void write(int)
void clear
IPerfStructElement next
void add(IPerfStructElement)
<<Interface>>
IElementSelector
IPerformanceStructureElement getNextElement
Operation Performed On Elements
The Structure To Be Benchmarked
Element Iteration Function
RunnableListMutator
head : IPerformanceStructureElement
elementsToAdd: long
fullIterations : int
operator: IListElementOperator
elementSelector: IElementSelector
builder : PerformanceStructureBuilder
run
Main
Flags : EFlagType
FlagCount : int
runBenchmark
Creates Initial Performance Structure Based On Flags
Performance Structure Builder
type : EPerformanceStructureType
IPerformanceStructure build()
Random Helper Class (so we plug in expected numbers for testing random based features)
Also standardises method of random selection
<<Interface>>
IRandRetriever
int getIntegerInRange(int min, int max)
Implements
Implements
RandRetriever
DeterministicRandRetriever
int expectedNumber

Figure 5 : GarbageBench UML Diagram.

Code Repositories Metrics

Code Metrics

Table 2 shows the code metrics extracted from our code repositories using git, as well as the location and version number of each of the software artifacts/deliverables presented in this section (summarized in Table 1).

Table 2 : Code Metrics and location of delivered SAs.

No	Software Artifact	Code Metrics (git)	Location	Release No
1	Transition of MaxineVM to Java8	29 commits 1257 LOC	https://github.com/beehive-lab/Maxine-VM/releases/tag/v2.2.0	v2.2.0
2	Continuous Integration (CI) Framework	22 commits 214 LOC	https://github.com/beehive-lab/Maxine-VM/blob/master/Jenkinsfile	-
3	AArch64 port of MaxineVM	457 commits 31868 LOC	https://github.com/beehive-lab/Maxine-VM/releases/tag/v2.4.0	v2.4.0
4	Integration with MMTk	78 commits 6914 LOC	-	-
5	GarbageBench	30 commits 2477 LOC	-	v0.1.0
Summarized code metrics		616 commits 42730 LOC		

Github Metrics

Figure 6 and Figure 7 are screenshots of the github statistics of the MaxineVM repository (https://github.com/beehive-lab/Maxine-VM) for the weeks of the v2.2.0 and v2.4.0 release respectively.

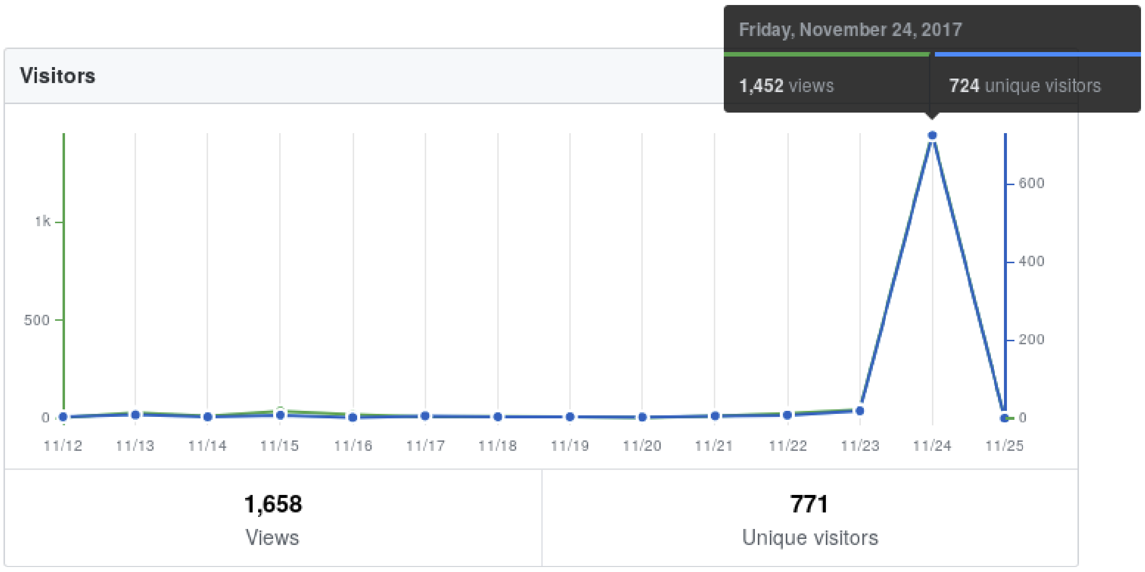


Figure 6 : Screenshot after MaxineVM's v2.2.0 release (23/11/2017) with Java 8 support.

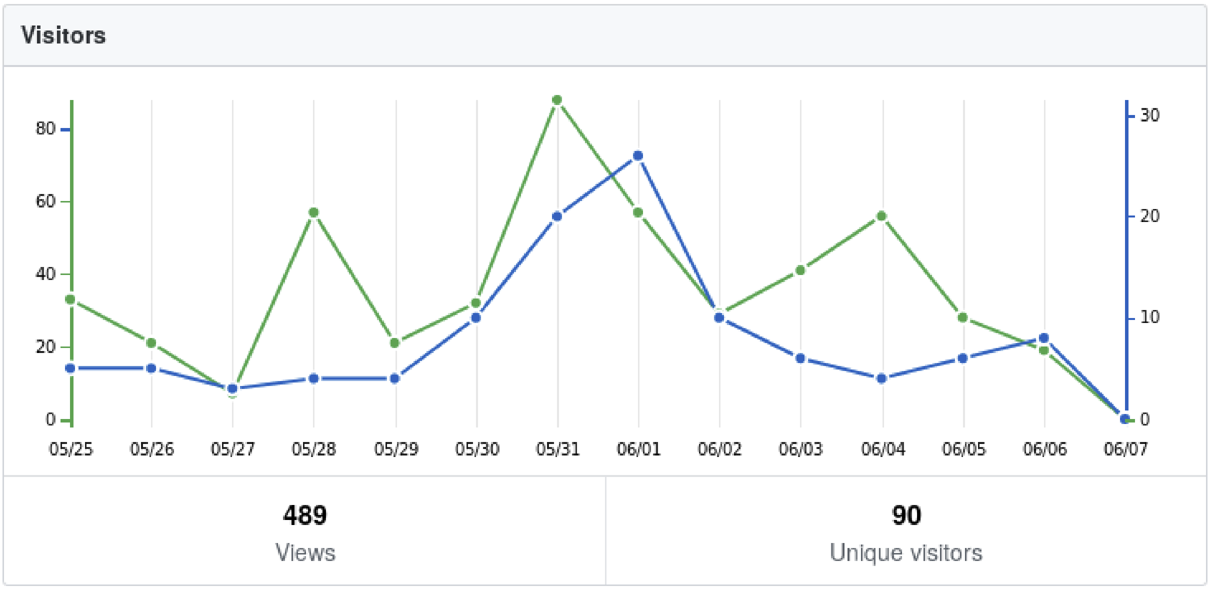
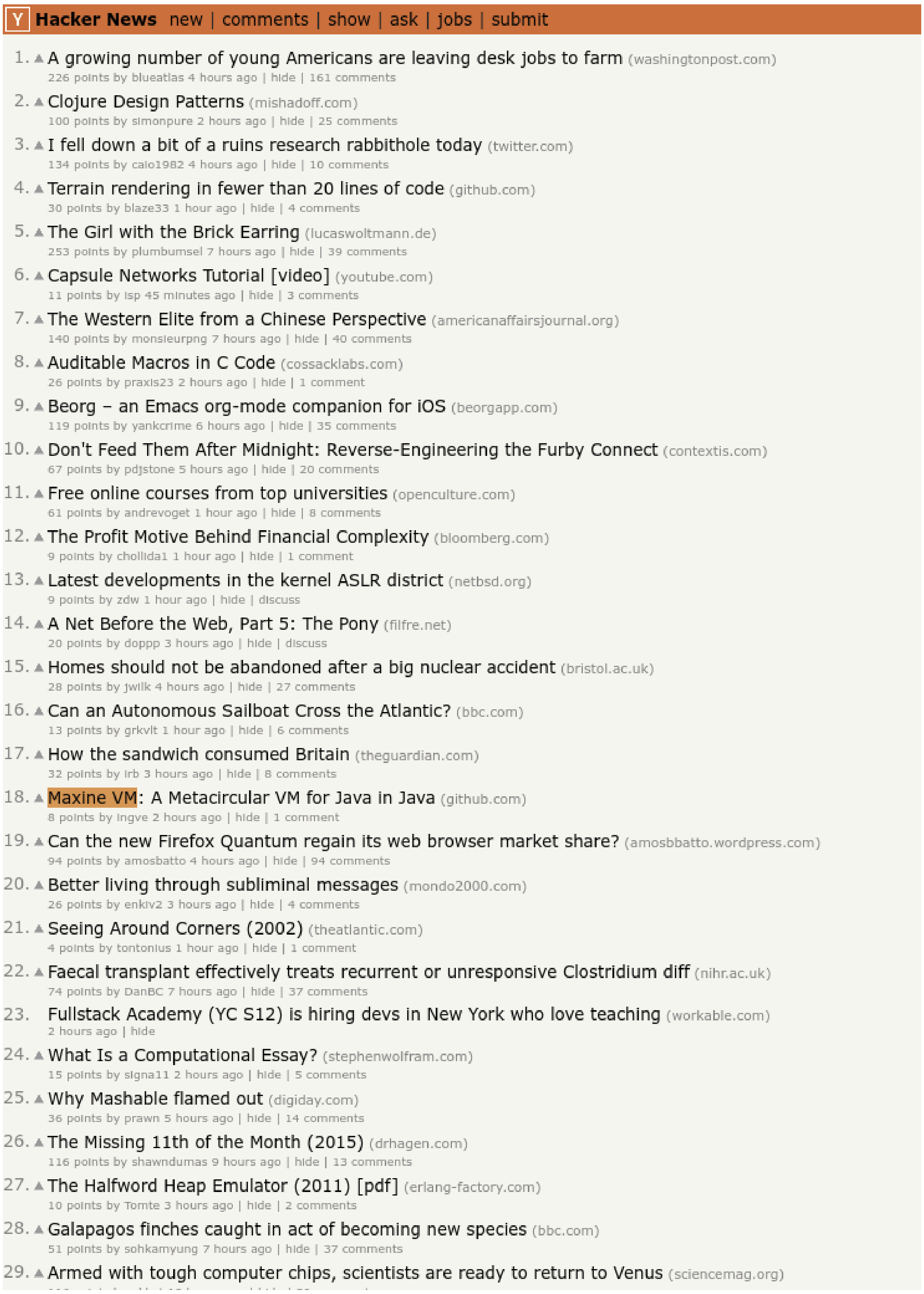


Figure 7 : Screenshot after MaxineVM's v2.4.0 release (30/5/2018) with HelloWorld on AArch64.

As is evident by Figure 6, one day after the release of v2.2.0 there was a rapid increase in the project's page that faded out next day. We attribute this behavior to the fact that that same day a post about Maxine VM had made it to the front page of "Hacker News" (see Figure 8) for 6 consecutive hours. Hacker News (https://news.ycombinator.com/) is one of the most popular social news website about computer science and technology.



Y **Hacker News** new | comments | show | ask | jobs | submit

1. ▲ A growing number of young Americans are leaving desk jobs to farm (washingtonpost.com)
   226 points by blueatlas 4 hours ago | hide | 161 comments
2. ▲ Clojure Design Patterns (mishadoff.com)
   100 points by simonpure 2 hours ago | hide | 25 comments
3. ▲ I fell down a bit of a ruins research rabbithole today (twitter.com)
   134 points by caio1982 4 hours ago | hide | 10 comments
4. ▲ Terrain rendering in fewer than 20 lines of code (github.com)
   30 points by blaze33 1 hour ago | hide | 4 comments
5. ▲ The Girl with the Brick Earring (lucaswoltmann.de)
   253 points by plumbumsel 7 hours ago | hide | 39 comments
6. ▲ Capsule Networks Tutorial [video] (youtube.com)
   11 points by isp 45 minutes ago | hide | 3 comments
7. ▲ The Western Elite from a Chinese Perspective (americanaffairsjournal.org)
   140 points by monsieurpng 7 hours ago | hide | 40 comments
8. ▲ Auditable Macros in C Code (cossacklabs.com)
   26 points by praxis23 2 hours ago | hide | 1 comment
9. ▲ Beorg – an Emacs org-mode companion for iOS (beorgapp.com)
   119 points by yankcrime 6 hours ago | hide | 35 comments
10. ▲ Don't Feed Them After Midnight: Reverse-Engineering the Furby Connect (contextis.com)
    67 points by pdjstone 5 hours ago | hide | 20 comments
11. ▲ Free online courses from top universities (openculture.com)
    61 points by andrevoget 1 hour ago | hide | 8 comments
12. ▲ The Profit Motive Behind Financial Complexity (bloomberg.com)
    9 points by chollida1 1 hour ago | hide | 1 comment
13. ▲ Latest developments in the kernel ASLR district (netbsd.org)
    9 points by zdw 1 hour ago | hide | discuss
14. ▲ A Net Before the Web, Part 5: The Pony (filfre.net)
    20 points by doppp 3 hours ago | hide | discuss
15. ▲ Homes should not be abandoned after a big nuclear accident (bristol.ac.uk)
    28 points by jwilk 4 hours ago | hide | 27 comments
16. ▲ Can an Autonomous Sailboat Cross the Atlantic? (bbc.com)
    13 points by grkvlt 1 hour ago | hide | 6 comments
17. ▲ How the sandwich consumed Britain (theguardian.com)
    32 points by lrb 3 hours ago | hide | 8 comments
18. ▲ Maxine VM: A Metacircular VM for Java in Java (github.com)
    8 points by ingve 2 hours ago | hide | 1 comment
19. ▲ Can the new Firefox Quantum regain its web browser market share? (amosbbatto.wordpress.com)
    94 points by amosbatto 4 hours ago | hide | 94 comments
20. ▲ Better living through subliminal messages (mondo2000.com)
    26 points by enkiv2 3 hours ago | hide | 4 comments
21. ▲ Seeing Around Corners (2002) (theatlantic.com)
    4 points by tontonius 1 hour ago | hide | 1 comment
22. ▲ Faecal transplant effectively treats recurrent or unresponsive Clostridium diff (nihr.ac.uk)
    74 points by DanBC 7 hours ago | hide | 37 comments
23. Fullstack Academy (YC S12) is hiring devs in New York who love teaching (workable.com)
    2 hours ago | hide
24. ▲ What Is a Computational Essay? (stephenwolfram.com)
    15 points by signa11 2 hours ago | hide | 5 comments
25. ▲ Why Mashable flamed out (digiday.com)
    36 points by prawn 5 hours ago | hide | 14 comments
26. ▲ The Missing 11th of the Month (2015) (drhagen.com)
    116 points by shawndumas 9 hours ago | hide | 13 comments
27. ▲ The Halfword Heap Emulator (2011) [pdf] (erlang-factory.com)
    10 points by Tomte 3 hours ago | hide | 2 comments
28. ▲ Galapagos finches caught in act of becoming new species (bbc.com)
    51 points by sohkamyung 7 hours ago | hide | 37 comments
29. ▲ Armed with tough computer chips, scientists are ready to return to Venus (sciencemag.org)

Figure 8 : Screenshot of Hacker News the next day of MaxineVM's v2.2.0 release (24/11/2017).

On less special circumstances, like in the case of the v2.4.0 release that didn't coincide with MaxineVM being in the front page of some major news platform, we observe three times more visits than usual. These numbers show that the community follows our progress and is interested in new releases of MaxineVM.

Publications

In parallel with the software development we have worked on publishing parts of our work in relative journals, conferences and workshops. Our efforts have resulted in the following publications:

1. Foivos S. Zakkak, Andy Nisbet, John Mawer, Tim Hartley, Nikos Foutris, Orion Papadakis, Andreas Andronikakis, Iain Apreotesei, Christos Kotselidis: **On the Future of Research VMs: A Hardware/Software Perspective.** MoreVMs 2018
2. Andrey Rodchenko, Christos Kotselidis, Andy Nisbet, Antoniu Pop, Mikel Luján: **Type Information Elimination from Objects on Architectures with Tagged Pointers Support**. IEEE Trans. Computers 67(1): 130-143 (2018)
3. Andrey Rodchenko, Christos Kotselidis, Andy Nisbet, Antoniu Pop, Mikel Luján: **MaxSim: A simulation platform for managed applications**. ISPASS 2017: 141-152
4. Christos Kotselidis, Andy Nisbet, Foivos S. Zakkak, Nikos Foutris: **Cross-ISA debugging in meta-circular VMs**. VMIL@SPLASH 2017: 1-9
5. Colin Barrett, Christos Kotselidis, Foivos S. Zakkak, Nikos Foutris, Mikel Luján: **Experiences with Building Domain-Specific Compilation Plugins in Graal**. ManLang 2017: 73-84
6. Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, Mikel Luján: **Heterogeneous Managed Runtime Systems: A Computer Vision Case Study**. VEE 2017: 74-82

All publications made in conferences and workshops were also presented in the corresponding venue, further disseminating our work in the ACTiCLOUD project.

4 Documentation

MaxineVM Build and Run Instructions

For detailed instructions on how to build and run the MaxineVM please follow the guide in the public wiki of the project:
https://github.com/beehive-lab/Maxine-VM/wiki/Build-and-Usage-Instructions

After the MaxineVM is successfully built, to run a Java application use `mx vm` instead of `java`, e.g. `mx vm -jar myApp.jar` or `mx vm HelloWorld`

Debugging using MaxineVM's Cross-ISA testing infrastructure

We have recorded a presentation of how MaxineVM's Cross-ISA testing infrastructure can be utilized to develop and debug back-ends for new ISAs. The recording can be found here: https://youtu.be/K-BZpAX_dvY