ACTiCLOUD: ACTivating resource efficiency and large databases in the CLOUD

Project No: 732366

H2020-ICT-2016-1

D3.6: Hyperscale JVM v2.0

Due date of deliverable:	**M32 (2019/08/31)**
Actual submission date:	M33 (2019/09/13)

Executive summary:

Deliverable D3.6 provides the final version of ACTiCLOUD's custom Java Virtual Machine implementation, named as Hyperscale JVM. The deliverable consists of the software source code and its documentation, with special focus on the tools developed during the project for research on memory management.

List of authors:

Author	Affiliation
Christos Kotselidis	UNIMAN
Foivos Zakkak	UNIMAN

Dissemination Level	**X**	**PU (Public)**
		PP (Restricted to other programme participants)
		RE (Restricted to a group specified by the consortium)
		CO (Confidential, only for members of the consortium)
	Where restricted, access granted to:	
Nature		R (Report)
		P (Prototype)
		D (Demonstrator)
	X	**O (Other)**

Review Status		Draft
		WP Leader accepted
		QA approved
	X	**Coordinator accepted**

Revision History:

Version	Author(s) (Affiliation)	Notes
0.1	Christos Kotselidis (UNIMAN)	Initial table of contents
0.2	Christos Kotselidis (UNIMAN) Foivos Zakkak (UNIMAN)	1st draft
0.3	Christos Kotselidis (UNIMAN) Foivos Zakkak (UNIMAN)	2nd draft
0.7	Atle Vesterkjaer (NSCALE) Jim Webber (NEO)	1st Internal review
0.8	Foivos Zakkak (UNIMAN)	Addressed reviewer comments
0.9	Vasileios Karakostas (ICCS)	2nd Internal review
1.0	Foivos Zakkak (UNIMAN)	Final version

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

Confidentiality:

Table of Contents

Figures

Tables

List of Abbreviations and Acronyms

Abbreviation / Acronym	Meaning
CI	Continuous Integration
CSP	Cloud Service Provider
DoA	Description of the Action
GC	Garbage Collection
HJVM	Hyperscale JVM
ISA	Instruction Set Architecture
JIT	Just-In-Time
JVM	Java Virtual Machine
LOC	Lines of Code
NUMA	Non-Uniform Memory Access
QA	Quality assurance
SA	Software Artefact
SO	Strategic Objective
WP	Work Package
WPL	Work Package Leader

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 deliverable, the HyperScale JVM is presented along with the technical outcomes of ACTiCLOUD with respect to Task T3.2 of WP3.

In the rest of this section we revise the task and the deliverable descriptions related to Task 3.2 "Hyperscale managed runtimes" as well as describe how the presented work relates to the project's Strategic Objectives and use cases of the partners.

1.1 Task 3.2: Hyperscale managed runtimes

Task 3.2: "Hyperscale managed runtimes" focuses on studying and understanding what optimizations can be applied to hyperscale managed runtime system. The task focuses on compilation, scheduling, and garbage collection optimizations within hyperscale managed runtime systems. Task 3.2 and in general the work undertaken in the context of HyperScale JVMs is based on OpenJDK's compilers (Graal). Graal is a state-of-the-art JIT compiler used by both industrial and research VMs. While, a part of the work is 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 1.2 and 2 describe in detail the necessary steps to achieve HJVM as well as our progress between M18 and M32. The resulting HJVM enables 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 is the first JVM capable of such functionalities.

1.2 In this Deliverable

The dual approach that we follow on ACTiCLOUD regarding the research and development of HJVMs has generated a number of software artefacts (SAs) 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 Artefact	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 Java 8 (previously on Java 7).	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.
6	Memory Management Profiler	Implementation of fine grained memory management profiling infrastructure in MaxineVM that integrates with NSCALE's performance monitoring tools	SO 1, 2	Necessary to identify and understand memory management patterns in order to implement memory optimizations for all use cases and platforms.
7	BCC-Java	eBPF based tracing tool that that relates changes in microarchitecture performance counter values to the execution of individual JVM system and application threads at low overhead	SO 1, 2	To identify if application threads suffer from load imbalance. To determine if blocking and lock contention overhead is significant.

As shown in Table 1, numerous developments have taken place in different fronts and contexts with the ultimate goal being the release of HJVM.

Figure 1, illustrates how the above software artefacts compose the resulting HJVM (described in more detail in Section 2).

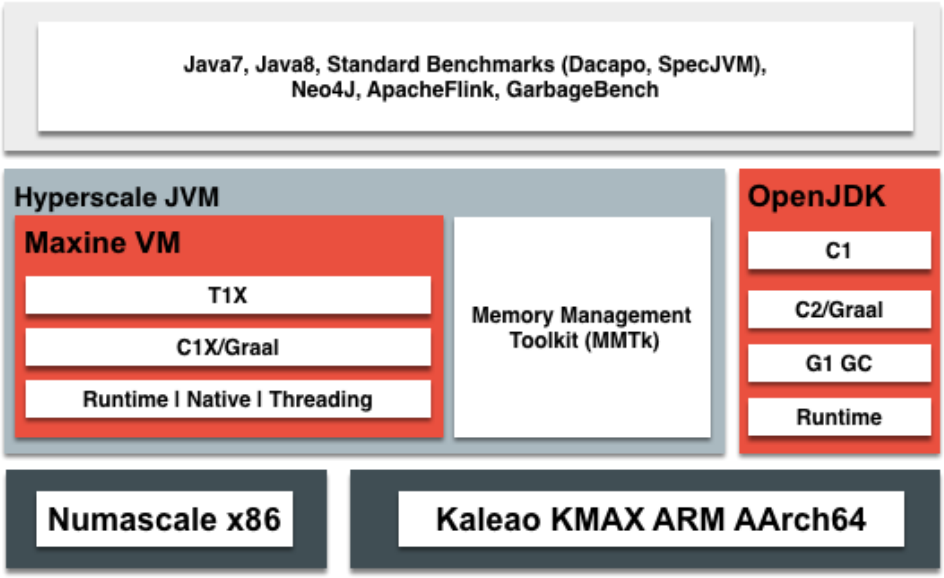


Figure 1: Relation between Software Artefacts (SAs) of Table 1 and the Hyperscale JVM (HJVM).

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 is capable of executing Java workloads, up to Java 8, including standard benchmarks, Neo4J, Apache Flink, and GarbageBench (SA 5).

1.3 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.

2 High-Level Description

Figure 1 depicts the Hyperscale JVM (HJVM) and its surrounding components. As shown, it consists of two main components: MaxineVM and MMTk. Both MaxineVM and MMTk were chosen over alternatives, e.g. OpenJDK HotSpot, due to their modularity and ease of extension. Both components are fully modular, allowing different algorithms to be used interchangeably in various segments of the runtime, like the JIT-compiler, the GC algorithms, etc. Additionally, both components are mainly written in Java easing the development of new components/modules. It is worth noting that despite not being based on the OpenJDK HotSpot VM, HJVM uses the OpenJDK standard java libraries and supports the OpenJDK Graal compiler. HJVM runs on both x86 and AArch64 architectures, essentially supporting both hardware platforms of the ACTiCLOUD architecture (Numascale-x86 and Kaleao KMAX-AArch64). In addition to the above, HJVM is able to run on the hypervisors provided by OnAPP. Finally, HJVM is able to execute the workloads of interest to ACTiCLOUD; namely Neo4J, Apache Flink and standard Java benchmarks.

2.1 Initial Status

In order to reach a fully functional state of the HJVM shown in **Figure 1**, the software artefacts (steps) of Table 1 had to be completed. The starting point of our work was MaxineVM running Java applications up to Java 7 only on the x86 architecture. Evidently, a significant development effort had to be made in order to transition to HJVM v1.0 and later to the current HJVM v2.0.

2.2 Hyperscale JVM v1.0 Status

In HJVM v1.0 we completed the following milestones:

1. **Java 8 support**
   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 programming language) as described in our paper published this year at the MoreVMs workshop where we set ACTiCLOUD's vision regarding future research VMs[1].
2. **Continuous Integration (CI) Infrastructure**
   Proper CI tools based on the Jenkins framework[2], 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.
3. **AArch64 initial support**
   With the help of the CrossISA Toolkit[3] we managed to successfully port the entire VM to the AArch64 architecture being able to execute HelloWorld on an AAarch64 odroid-c2

[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***. In MoreVMs 2018 Workshop on Modern Language Runtimes, Ecosystems, and VMs, April 2018.
[2] https://jenkins.io
[3] 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.

board. The successful bootstrapping and execution of the HelloWorld example of MaxineVM on AArch64, was 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. HJVM v1.0 achieved the following satisfactory pass rates, which we improve in HJVM 2.0:

- JTT Tests (~3000): ~99.7%
- SPECjvm2008 (single threaded): ~84%
- DaCapo (single threaded): ~54%

4. **MMTk initial integration**
   MMTk[4] is a collection of GC algorithms developed initially in the context of JikesRVM[5]. MMTk offers not only a set of high performing GC algorithms but it also provides the basic building blocks for implementing new ones, thus making it a valuable tool for GC research on the HJVM. In HJVM v1.0 we completed the integration of MMTk in MaxineVM's building process as well as the unification of the addressing types of MMTk and MaxineVM.
5. **GarbageBench**
   In HJVM v1.0 we released a pre-alpha release of GarbageBench which 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 patterns and behaviours found in real-world applications. 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 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 or set of data structures to be used during execution. This way, we can simulate different workloads and access partners. For example, parts of Neo4J (a graph database) like accessing nodes and relationships via graph traversals can be simulated by using the graph data structure of GarbageBench.

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

2.3 Summary of Improvements over HJVM v1.0

Since HJVM v1.0 (M18) UNIMAN has released 6 public releases of MaxineVM (the core JVM of HJVM). These releases, among others enable HJVM to:

- Use the latest Java 8 libraries from OpenJDK
- Run more benchmarks and applications
- Profile memory allocations performed by applications
- Run faster, enabling the parallel JIT compilation of methods
- Run Java 8 code faster, by supporting jsr292 (Java 8 features, such as lambdas) in the optimizing compiler (C1X)
- Be built and ran inside a docker container (allowing it to be more reliably tested and easily adopted)
- Be built and ran on the Mac OS (allowing cross-platform development and testing)

In the following sections we describe in more detail the key features added in HJVM v2.0 along with external tools that enable research for NUMA-aware memory management.

3 HJVM memory profiler

The HJVM profiler (integrated since release 2.7[6]) enables the inspection of allocation patterns of applications running on the HJVM. The profiler records the size and type of each object along with the ID of the thread that allocates it and the ID of the NUMA memory location at which it allocated. Such information is necessary to understand the behaviour of different applications on NUMA machines and design NUMA-aware garbage collection algorithms and memory allocators.

Profiling all memory allocations of an application incurs significant overhead. Furthermore, as is standard with measurements in managed runtime systems, warm up measurements are usually very noisy and not representative of steady state. To enable fine control of the profiling window, HJVM implements three different policies, as discussed next.

3.1 Flare policy

The Flare policy allows the user to define a *flare* class. Whenever an allocation happens, HJVM checks whether the allocated object is an instance of the flare class and depending on the configuration parameters enables or disables profiling. More specifically, the Flare policy can be configured to start the profiling after N allocations of flare class instances and keep profiling till another M instances of the flare class are allocated. This policy is useful in cases where a special object is allocated just before performing a procedure of interest. For instance, this can be a class describing an input request in a server application. With the Flare policy we can instruct HJVM to ignore the first N requests and profile the following M requests.

3.2 Explicit GC policy

The Explicit GC policy is similar to the Flare policy, but enables profiling after N explicit GCs -- invocations of `System.gc()`-- and keeps profiling enabled for the next M explicit GCs. This policy is handy for profiling benchmarks that run inside harnesses, e.g. the DaCapo benchmark suite, that explicitly perform garbage collection before each benchmark phase to ensure similar behaviour across phases. With the Explicit GC policy we can instruct HJVM to ignore the first N phases and profile the following M phases, allowing us to focus on the more stable phases of a benchmark.

3.3 Explicit control-flow policy

The explicit control-flow policy is the most advanced among the three. With the explicit control-flow policy we can define a method as an entry-point and another method (or the same) as an exit-point. Then, HJVM enables profiling when the entry-point method gets invoked for the first time and disables profiling when the exit-point method gets invoked. Note that if the entry-point method gets invoked again, profiling is enabled again. With the Explicit control-flow policy we can focus on a very specific segment of an application and inspect its behaviour.

3.4 NUMASCALE performance counters monitoring integration

HJVM memory profiler can be combined with the `vmxstat` and `numascope` tools[7], described in Deliverable 3.5: "ACTiCLOUD-enabled system libraries v2.0", to better understand the behaviour

[6] https://github.com/beehive-lab/Maxine-VM/releases/tag/v2.7.0
[7] https://github.com/numascale/numascope

of applications and their performance. The HJVM profiler provides insight from the JVM's perspective, while the NUMASCALE performance counters provide insight from the machine's perspective. Observing only one side may lead to incorrect conclusions. For instance, the HJVM profiler might report many remote allocations and remote accesses, hinting that there is a locality issue in the application that needs to be optimized. Inspecting the hardware counters however may show a different story, e.g. that most of the remote accesses are being cached and they do not induce a big overhead to the application, thus there is no need to optimize them.

HJVM profiler is fine grain, while the `vmxstat` and `numascope` tools sample a process on predefined sample periods. To combine the two and obtain hardware metrics that correspond to the specific part of the application that we are profiling, we annotate the `numascope` graphs by writing labels to `/run/numascope-label` each write to this file produces a timestamped mark on the graph allowing us to understand when a specific operation started and completed, e.g. a garbage collection cycle.

3.5 Memory Analysis

Using the HJVM memory profiler we perform an analysis on the allocation patterns of the DaCapo benchmark suite and Neo4J. The analysis' results help us identify imbalances in the way memory is allocated and pinned to NUMA-nodes.

3.5.1 DaCapo

The DaCapo benchmark suite[8] is a collection of Java applications used to evaluate the performance of JVMs, mainly focusing on memory management. The DaCapo benchmark suite is widely adopted by the Java community and its set of applications are considered representative of real Java workloads. However, the current version of the benchmark suite (9.12-bach) was not meant to scale to hundreds of CPUs, nor to TBs of main memory. To remedy this, the maintainers of the benchmark suite are preparing a new release with newer and more scalable Java applications. That said, until a stable release of the new version becomes publicly available we cannot utilize all the resources of the KMAX or the NUMSCALE platform. Consequently, we chose to run the DaCapo suite and profile its applications on a smaller machine with 2 NUMA-nodes, 8 cores (16-threads) and a total of 380 GB main memory.

The memory analysis of the applications indicates that different applications exhibit different behaviour. There are applications (e.g. fop) that perform the majority of the allocations from a single thread, using the memory in the NUMA-node that that thread resides at. There are balanced applications (e.g. `xalan` and `lusearch`) that perform about the same amount of allocations from each thread in the execution, and end up spreading the data evenly across both the NUMA-nodes. There are also applications (e.g. `pmd`) that perform a large portion (50%) of the allocations from a single thread, and the rest are evenly distributed between the other threads. In the last case the memory of *dominant* thread's NUMA-node is mostly used, resulting in unbalanced memory usage.

Based on the above observations we create the following categories of applications:

I. **Initializer**: A single thread, the *initializer*, performs most of the allocations and then distributes work to other threads that do not perform further allocations and consume the data created by the *initializer* thread.

[8] http://dacapobench.sourceforge.net

II. **Dominated**: Although all threads perform allocations, there is a single thread, the *dominant thread*, (or a small group of threads) that performs a large amount of the allocations. This dominance results in a large amount of data to reside on a single NUMA-node, that of the dominant thread, resulting in unbalanced data distribution.
III. **Balanced**: All threads perform about the same number of allocations, resulting in a balanced distribution of the data across the NUMA-nodes.

3.5.2 Neo4J

Neo4J is one of the project's use cases that utilizes the JVM as its execution engine. As per Neo4J's documentation[9] Neo4J relies on the JVM's heap for its dynamic memory needs. To avoid reading from the disk, Neo4J also employs a page cache. This page cache resides outside the JVM heap to avoid the additional overhead it would incur to garbage collection. That said HJVM (or any JVM for that matter) cannot optimize the performance of the page cache, instead there is the potential to optimize the management and placement of the dynamic memory allocations of Neo4J. Neo4J showcases why the HJVM profiler is important and how it differs from the performance monitoring tools provided by NUMASCALE. If we were to profile Neo4J using only the performance monitoring tools provided by NUMASCALE it would be impossible to distinguish between the accesses to the page cache and those to the Java heap. HJVM gives us a better understanding of what is happening on the Java heap and in combination with the performance monitoring tools of NUMASCALE it can give us the complete picture.

We use the HJVM profiler to observe the allocation patterns of Neo4J for a set of queries in the Linked Data Benchmark Council (LDBC) - Social Network Benchmark (SNB)[10]. Our experiments are based on the Neo4J community edition v3.5.4, which does not support executing queries in parallel, i.e. breaking the queries to operators that may be executed on different threads. It allows however multiple queries to be executed concurrently (one per thread). In our experiments we perform a different run for each query of LDBC-SNB separately. For each query we run 100 instances of the corresponding query as a warmup, then run 50 more and profile the next one. We run our experiments on the NUMSCALE system setup in Athens. We configure Neo4J to use 4 threads in total; pinning the first two on cores 0 and 2, and the second two on cores 4 and 6, essentially using two NUMA-nodes. The dataset we use is the SF100, which consists of more than 100GB data.

[9] https://neo4j.com/docs/operations-manual/current/performance/memory-configuration
[10] http://ldbcouncil.org/benchmarks/snb

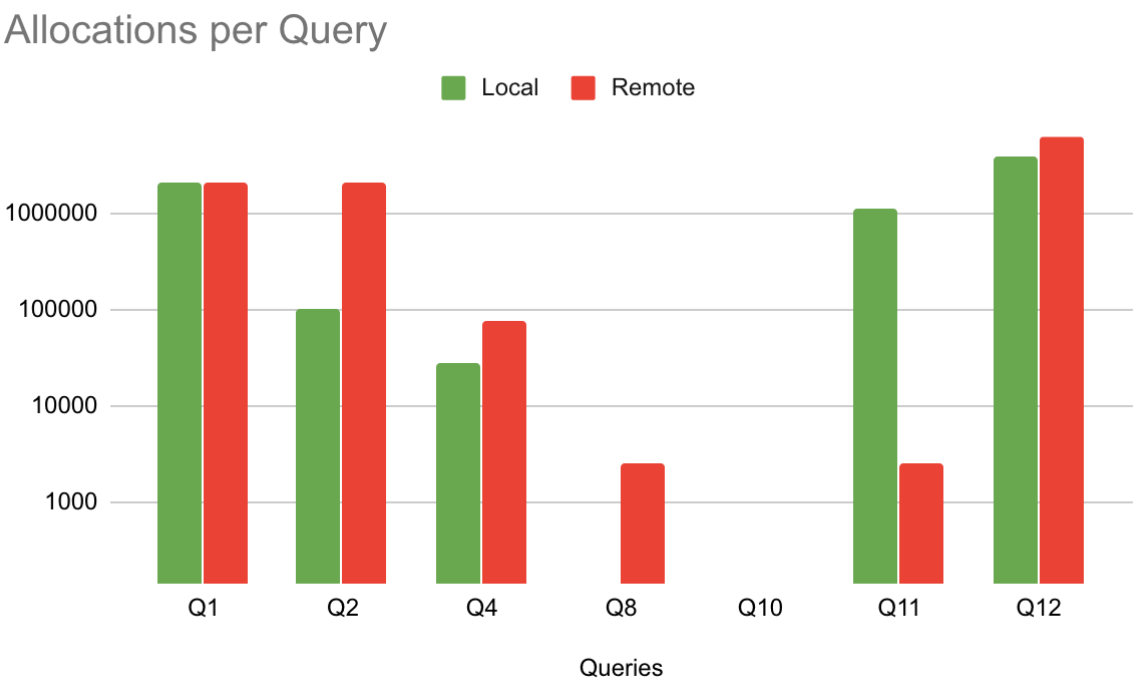


Figure 2: Local vs Remote Object Allocations per Query

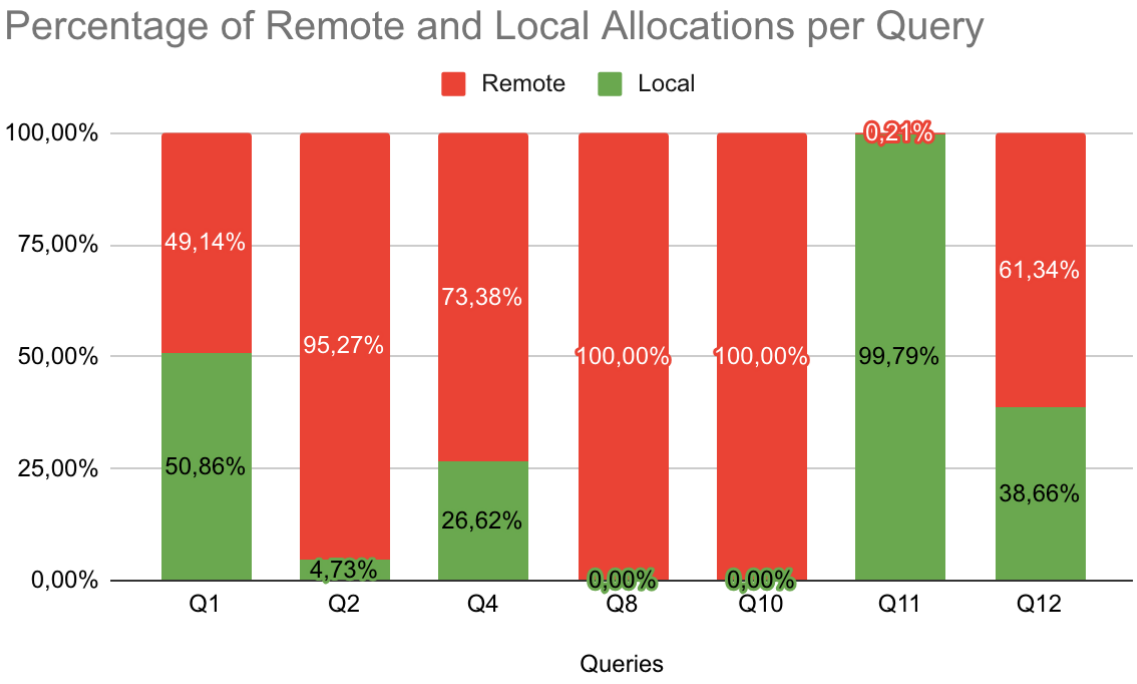


Figure 3: Percentage of Remote and Local Allocations per Query

Figure 2 and **Figure 3** present the results. **Figure 2** illustrates the number of allocations performed on the local and the remote NUMA-node for each query. **Figure 3** illustrates the percentage of remote and local NUMA-node allocations per query. We observe that different queries have different memory requirements, and thus allocation patterns. Queries that perform

less allocations tend to allocate all memory on a single node (which happens to be the remote NUMA-node in our measurements). On the other hand, queries that perform millions of allocations spread the allocations on both NUMA-nodes despite not requiring enough memory to justify this behaviour. We plan to further investigate Neo4j's allocation behaviour to better understand why the above patterns occur.

3.5.3 Optimization

HJVM v2.0 enables us to combine the allocation profiler results with performance counters obtained using NUMASCALE's performance counter monitoring tools. The combined metrics will allow us to better understand the relationship between the application category (as described above) and the number of remote accesses performed by the application. This will allow us to better understand where in the JVM NUMA-awareness matters the most. Is it during allocation, during the moving phase of garbage collection, etc.?

Below we list a set of potential optimizations and how they could impact the application categories discussed above:

I. **Allocation Shuffling**: In many applications (especially those in the **Initializer** category) a thread allocates objects that will then be used by other threads. For instance, a thread opens and reads a file to create a data-structure that will then be processed in parallel. In NUMA systems this typically results on all data residing in a single NUMA-node and then the threads that process that data to pay the cost of remote memory accesses.

   To avoid this, the JVM can shuffle allocations on the available NUMA-nodes. For instance, the `numactl` tool supports the `interleave` memory allocation policy. This policy dictates that each allocation will happen on a different NUMA-node in a round-robin fashion.

   A disadvantage of this policy is that it is totally application agnostic. It might end up allocating data that will be processed by the same thread on different NUMA-nodes, or even worse to allocate data that are always accessed one after the other on different NUMA-nodes, not allowing them to be efficiently prefetched in the caches.

   This kind of optimization is expected to reduce remote memory accesses for applications that fall in the **Initializer** category, but not for the rest.

II. **NUMA-aware GC moves**: Most GC algorithms, are moving the data on garbage collection. This property gives the JVM the opportunity to relocate objects to different NUMA-nodes as well. To take advantage of this, the JVM needs to profile memory accesses at runtime to figure out what threads are accessing what objects the most and try and bring them together by moving the objects to the NUMA-nodes of the corresponding threads.

   The disadvantages of this approach are that it requires profiling memory accesses, the overhead of which might be prohibitive. Additionally, it complicates the logic of the GC algorithm which might result in longer GC pauses.

   This approach is expected to reduce remote memory accesses in all application categories.

III. **NUMA-aware thread scheduling**: Similarly, to moving objects at GC the JVM can also move threads at runtime instead. That said, a thread running on a different NUMA-node than the one holding the data it needs to access can be moved to the desired NUMA-node to reduce remote memory accesses.

The disadvantages of this approach is that it requires profiling (or analysis of the code) to understand which data a thread is going to access and where these data reside at. Furthermore, moving threads will possibly negatively impact the effect of hardware caches, since when moving to a different core, the hardware cache will be cold. Additionally, this approach requires data to be evenly distributed across NUMA-nodes, otherwise threads will end up competing for the cores of a single (or a few) NUMA-node(s).

This approach is expected to reduce remote memory accesses in applications falling in the **balanced** category, since it depends on a balanced distribution of the data.

Ideally, the JVM should provide sane defaults that would not favour a single category, but it should also allow tuning for specific categories through parameters. That said we believe that the most promising approach is that of the **NUMA-aware GC moves**.

4 BCC-Java

Additionally, to the HJVM profiler we have developed a JVM-independent low-overhead (overhead geomean 3.6%) tracing tool that records microarchitectural and execution time characteristics associated with the multi-threaded execution of Java programs. The BCC-Java tool exploits the *extended Berkeley Packet Filter (eBPF)* instrumentation and tracing framework available as standard in 4.4+ Linux kernels. BCC-Java currently records a thread's name, its start/exit time, cycles, instructions, and time spent on-core in execution. Additionally, it records the number of scheduling quanta given to each thread, the off-core waiting time, and the total number of `sys_futex` system calls. The initial JVM process is created by executing the JDK `java` command; hereafter referred to as the java-process. Our tool uses instrumentation of:

- java-process creation and termination by instrumenting kernel methods for `sched_process_exec`, `sched_process_fork` and `sched_process_exit`. The PID of the initial java-process that is created is stored in a hashmap.
- The creation of new threads that are children of the initial PID are tracked using a kernel probe attached to the `sys_clone` method.
- Kernel probes are attached to the entry and exit of the `sys_futex` system call. We record for each thread, the number of `futex` calls, the total elapsed time whilst blocked, and performance counter values when a thread is blocked/unblocked.
- Thread scheduling, namely `sched_switch` is instrumented to collect time and performance counter information concerning the allocation of a thread to a processing core. The counter values and elapsed time are updated at the end of a thread's quantum, or when the thread enters the `sys_futex` system call.
- On termination of the java-process, the tool stores per-thread details of accumulated performance counter values, total wait time, total calls to `sys_futex`, and elapsed time.

BCC-java was developed to provide a useful low-overhead mechanism to determine if there are any thread-level performance issues that need further investigation using higher overhead tools and mechanisms. It traces all threads, i.e. both the application and the internal JVM system threads that provision important subsystem features, such as JIT compilation and Garbage Collection, that can be significant sources of overhead. The unique features of BCC-Java are that it does not require any modifications to the JVM, nor do applications need to be executed with special JVM arguments.

BCC-java reports per-thread microarchitecture counters that enable load-imbalance and blocking behaviour to be identified and related to thread names, that can in turn be related to the application and to specific JVM service/subsystems. Blocking and lock contention overheads, due to `sys_futex` system calls, can be analyzed to determine if they are a significant percentage (overhead) of the overall thread lifetimes. The overall context of the BCC-java tool, its relation to other profiling and tracing tools, and how it can be used to identify a range of specific performance problems are described in our ICPE '19 paper[11] that investigates the DaCapo benchmarks. For example, if blocking and locking behaviour are found to be a significant overhead, then the paper illustrates how the deployment of the BCC/eBPF *offwaketime* tracing tool (having higher execution overhead, and requiring the usage of additional JVM arguments such as -XX:+PreserveFramePointer) can be used to produce *flamegraph*[12] visualizations that

[11] Andy Nisbet, Nuno Miguel Nobre, Graham Riley, and Mikel Luján. 2019. ***Profiling and Tracing Support for Java Applications***. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering* (ICPE '19). ACM, New York, NY, USA, 119-126. DOI: https://doi.org/10.1145/3297663.3309677

[12] http://www.brendangregg.com/flamegraphs.html

directly relate a long-latency blocking thread call-stack to the call-stacks of threads that cause the blocked thread to wake up.

5 MMTk Integration

As described in *D3.2: Hyperscale JVM 1.0*[13], the integration of MaxineVM (the core JVM of HJVM) with MMTk will ease the development of new GC algorithms and therefore make HJVM more appealing for memory-related research. In HJVM v1.0 we had successfully performed some initial integration allowing MMTk to be build using MaxineVM's build process, and unifying the addressing types of the two projects. Further integrating MaxineVM with MMTk was a challenging process due to the large code-bases of both projects; collectively over 600K lines of code. To minimize the changes on both platforms, in HJVM v2.0 we implemented a new layer between the two, taking advantage of the modularity of both platforms. MaxineVM provides the *HeapScheme* interface which allows the seamless plug of different GC algorithms, as long as they implement the interface. As shown in **Figure 4** we implement the *HeapScheme* interface for the MMTk framework. This way MMTk can be used as a replacement to the existing GC algorithms of MaxineVM, ultimately allowing any MMTk-based GC algorithm to be used with MaxineVM.

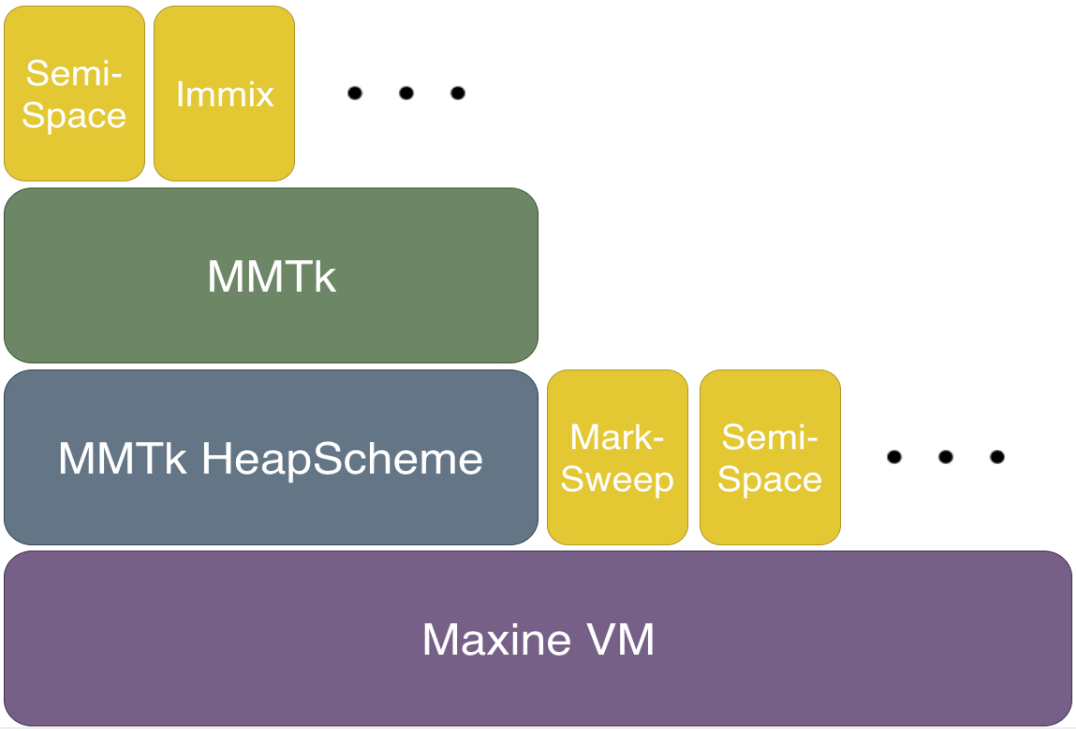


Figure 4: MaxineVM-MMTk integration through the MMTk HeapScheme

At its current state the integration supports the semi-space garbage collector and is considered to be in alpha stage. The integration code is not yet published on MaxineVM's public master branch, due to license issues that we are currently working to resolve. More specifically, MaxineVM is licensed under the GNU Public License v2 (GPL v2), while MMTk is licensed under the Eclipse Public License v1 (EPL v1). These two licenses are incompatible, so we are working with the MMTk team to make a new release of the MMTk source code under the Eclipse Public License v2.0 (EPL v2.0) with the Secondary License clause. This will essentially dual-license MMTk with EPL v2.0 and GPL v2.0+ which will make it compatible with MaxineVM.

[13] https://acticloud.eu/deliverables/uploads/tr/3.2

6 AArch64 port

As of HJVM v1.0 we have improved AArch64 support in HJVM by fixing bugs and optimizing performance. The two most important optimizations that we performed for v2.0 are the addition of jsr292 support to the C1X optimizing compiler and the optimization of patchable call-sites.

6.1 JSR292 support in C1X for AArch64

The Java Specification Request 292 (JSR292) introduces a new Java byte-code called `invokedynamic`. This bytecode enables JVMs to run dynamically-typed programming languages, by enforcing the type-checking at runtime instead of compile-time. To support this new byte-code, hardware-specific stubs are required. In HJVM v2.0 we have implemented those stubs for both x86-64 and AArch64 platforms mainly to improve performance of Java workloads that use lambdas, but also to better support dynamically-typed languages. Furthermore, since the introduction of lambdas in Java, standard Java libraries that are making use of them are increasing. That said, even applications without lambdas in their code are impacted by this contribution, as long as they use some standard Java library that uses lambdas.

6.2 Patchable call-sites

As we discuss in our paper to appear at the 16th International Conference on Managed Programming Languages & Runtimes (MPLR'19, formerly ManLang), managed runtime systems that employ Just-in-Time (JIT) compilation --especially those supporting tiered compilation-- require dynamically patchable call-sites. Such runtime systems self-modify their code in order to optimize it. Typically, this is happening at the method-level, i.e. a method gets optimized and all its callers are patched to invoke the freshly generated code. In x86-64 platforms call-sites are typically implemented with a single instruction, making patching a trivial process of altering a single instruction using an atomic instruction like compare and swap. In RISC (Reduced Instruction Set Computer) architectures such as AArch64, more than one instruction might be required to perform a long-range call, i.e. more than 128MB away from the current program counter value. Long-range calls allow for larger code-caches, thus for more optimized code and hence better performance overall. To make matters worse, AArch64 also restricts the instructions that can be safely patched while being executed. To find the best possible implementation that is sound, efficient, and compact in terms of code size, we performed a number of experiments with different patchable call-site implementations to select the best implementation during the HJVM's port to AArch64 (Kaleao KMAX).

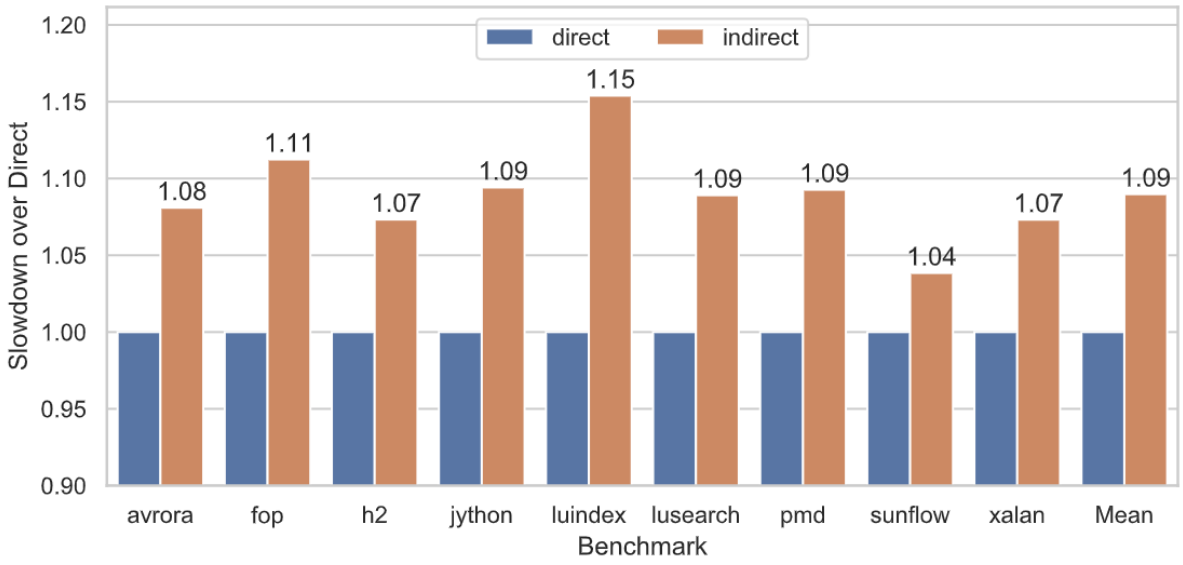
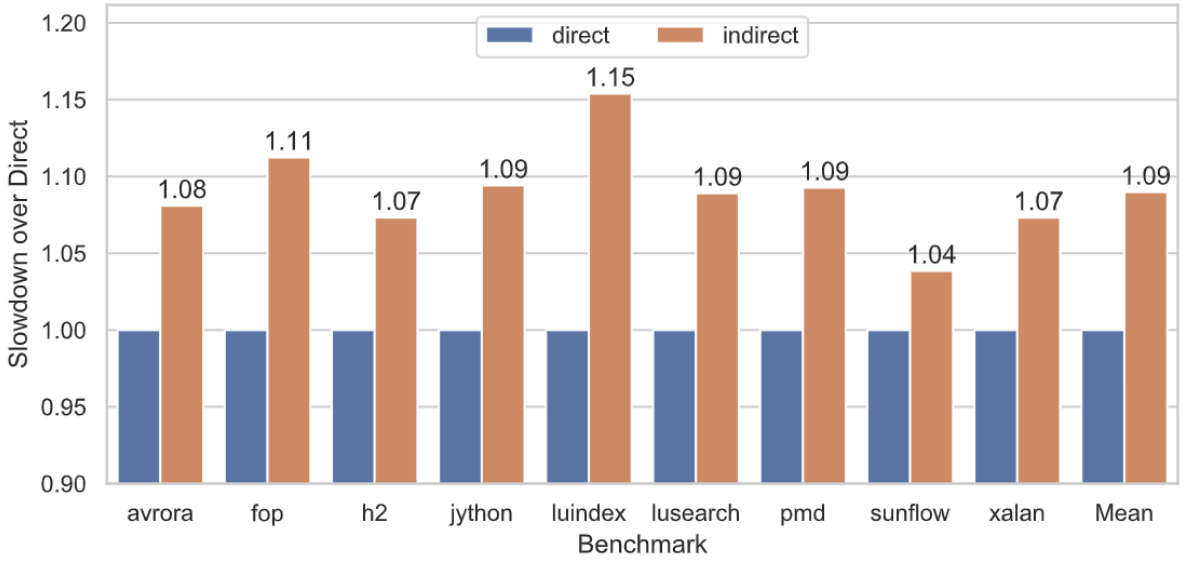


Figure 5: Performance impact of different call-site implementations

Our experimentation has indicated that the best performing implementation of long-range patchable call-sites has an overhead in the range of 4%-15% with a geometric mean of 9% as shown in



. We find this overhead significant and propose as an optimization the use of direct branches (single instruction) whenever possible, e.g. small applications where calls don't jump further than 128MB from the program counter.

6.3 Pass rates

In HJVM v2.0 we have also fixed a number of bugs resulting in higher pass-rates in the DaCapo benchmark suite. Specifically, we increase the pass-rate from 54% running on a single thread to 82% with multithreading enabled as well. This pass rate enables us to use HJVM in our experiments on AArch64, as we did for the call-site implementations. In the meantime, we continue to work on increasing HJVM's pass-rates on AArch64.

7 Code Repositories 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 publicly available software artefacts/deliverables presented in this deliverable. The presented code metrics are essentially the delta between HJVM v1.0 and HJVM v2.0.

Table 2: Code Metrics and location of publicly available delivered SAs

No	Software Artefact	Code Metrics (git)	Location	Release No
1	HJVM profiler	235 commits 2672 LOC[14]	https://github.com/beehive-lab/Maxine-VM/releases/tag/v2.8.0	v2.7.0 and v2.8.0
2	Parallel JIT compilation and faster Java 8 code	158 commits 11117 LOC	https://github.com/beehive-lab/Maxine-VM/releases/tag/v2.5.0	v2.5.0
3	AArch64 port improvements	81 commits 503 LOC	https://github.com/beehive-lab/Maxine-VM/releases/tag/v2.5.1	v2.5.1
4	Integration with MMTk	252 commits 11359 LOC	Not yet released	-
5	Continuous Integration	45 commits 437 LOC	https://github.com/beehive-lab/Maxine-VM/releases/tag/v2.7.0	v2.7.0
6	BCC-java	5 commits 570 LOC	Not yet released	-
Summarized code metrics		776 commits 26658 LOC		

[14] LOCs are calculated as the diff of the number of line insertions minus the number of line deletions, as obtained by `git log --shortstat`.

8 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:

- Andy Nisbet, Nuno Miguel Nobre, Graham Riley, and Mikel Luján: **Profiling and Tracing Support for Java Applications**. In ICPE 2019.
- Timothy Hartley, Foivos S. Zakkak, Christos Kotselidis, and Mikel Luján: **An Analysis of Call-site Patching Without Strong SMC Hardware Support.** To appear in MPLR 2019.

The publications are also presented in the corresponding venue, further disseminating our work in the ACTiCLOUD project.

9 Conclusions

This deliverable contains information about the successful implementation of HJVM v2.0 and its accompanying tools that, when combined, form a GC research platform that enables robust experimentation and profiling of memory management algorithms on Java applications.

UNIMAN will continue using the developed GC research platform to understand in more depth the allocation and memory access patterns of the ACTiCLOUD use-cases, mainly focusing on the Neo4J use-case which relies heavily on the JVM. The ultimate goal of these efforts is to further optimize the ACTiCLOUD use-cases. In the process new additions/extensions might be made to HJVM.

Appendix A: Documentation

A.1 MaxineVM Build and Run Instructions

For detailed instructions on how to build and run the MaxineVM please follow the instructions in the public documentation page of the project:

https://maxine-vm.readthedocs.io/en/stable/build.html

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`

A.2 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