Towards Practical Heterogeneous Virtual Machines
James Clarkson, Juan Fumero, Michail Papadimitriou, Maria Xekalaki, Christos Kotselidis
The University of Manchester
United Kingdom, M13 9PL
first.last@manchester.ac.uk

ABSTRACT
Heterogeneous computing has emerged as a means to achieve high performance and energy efficiency. Naturally, this trend has been accompanied by changes in software development norms that do not necessarily favor programmers. A prime example is the two most popular heterogeneous programming languages, CUDA and OpenCL, which expose several low-level features to the API making them difficult to use by non-expert users.

Instead of using low-level programming languages, developers tend to prefer more high-level, object-oriented languages typically executed on managed runtime environments. Although many programmers might expect that such languages would have already been adapted for execution on heterogeneous hardware, the reality is that their support is either very limited or totally absent. This paper highlights the main reasons and complexities of enabling heterogeneous managed runtime systems and proposes a number of directions to address those challenges.

CCS CONCEPTS
• Software and its engineering → Virtual machines;

KEYWORDS
Virtual Machines, Java, OpenCL, GPU, FPGA

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ACM Reference Format:

1 INTRODUCTION
Heterogeneous computing, initially introduced in the form of GPGPUs, has recently become very popular as a means to accelerate various applications from different domains including HPC and Big Data. Towards the transition to heterogeneous computing a number of new programming languages has emerged with CUDA and OpenCL being the most prevalent. Both languages enable the execution of high performance code on a variety of devices including GPUs and FPGAs, and have been designed at a lower level than non-expert programmers would expect. Both the prerequisite in-depth understanding of the architectural characteristics of the underlying heterogeneous hardware, and the exposure of low-level primitives to the API, make their use more challenging by developers, who typically write code in more widely used high-level programming languages such as Java, C#, Python, R, and JavaScript [14]. These languages typically run on top of a Managed Runtime Environment (MRE) and, in order to exploit a heterogeneous hardware resource (e.g. GPU), wrapper libraries are required to direct execution to pre-compiled OpenCL/CUDA kernels. Such an approach not only “violates” the semantics of the high-level programming language, but also requires from developers to gain expertise on new low-level programming environments.

Recently, a significant amount of work [1–9, 12, 13, 15] has been dedicated to enable dynamic JIT compilation of high-level code (e.g. Java, JavaScript, R) to heterogeneous hardware via OpenCL or PTX/CUDA. These approaches partially solve the challenge of heterogeneous execution since they mainly focus only on the aspect of compilation. MREs are complex systems encompassing a number of interconnected software components such as compilers, runtimes, memory managers, and garbage collectors. Consequently, all of these subsystems have to be re-engineered to enable performance and usability from interpreted programming languages. The remainder of this paper discusses the challenges of enabling heterogeneous execution of MREs and presents our proposal and work in progress towards addressing those challenges.

2 CHALLENGES
In this section we highlight the most significant challenges in designing heterogeneous MREs.

Programmability: High-level features of managed languages such as dynamic memory allocation and management, virtual calls and exception handling, although supported by Java and other languages, they are not present in CUDA or OpenCL. Therefore, they are not naturally supported on heterogeneous devices.

Transparency: Platform portability is a key design principle of Java that has been achieved through virtualization and abstraction of the underlying hardware architecture by the JVM. Similarly, the integration of hardware accelerators in the JVM should also follow the same principle. Developers should be able to run their code on the underlying device without having to explicitly manage memory, parallelism, code placement and other added complexities.

Adaptability: The existing compilers of JVMs have been tuned throughout the years for CPU execution exploiting Instruction Level Parallelism (ILP). Heterogeneous hardware accelerators, however, are built for different execution scenarios such as data parallelism (GPUs) or task pipelining (FPGAs). In addition, different
GPUs have different characteristics and capabilities that compilers must accommodate. Therefore, JVMs must be able to dynamically adapt the generated code based on the particular hardware device transparently to the user.

**Device Portability:** Breaking away from the norm that applications always execute on CPUs, heterogeneous MREs must accommodate device portability. For example, a vanilla Java application should be able to exploit any hardware device, even if unknown during development time. In addition, in cloud deployed Big Data applications where fault tolerance is important, MREs should be able to adapt the code (through de-optimization and re-compilation) if a node with a specific device fails.

**Performance Portability:** The main motivation behind hardware accelerators is increased performance. A heterogeneous MRE should aim in delivering the same performance when executing the code on different accelerators. The trade-offs between peak performance, compilation time, and heterogeneous execution should always yield the best possible performance regardless of the type of accelerator that exists on our system. Therefore, a more sophisticated decision-making model is required compared to the existing counter-based compilation of "hot" methods.

### 3 PROPOSAL

This section outlines our proposal for heterogeneous MREs with some initial results described in [10, 11].

**Task-Based API.** To address programmability, we propose a task-based API for heterogeneous programming that augments existing Java APIs in a seamless manner. Developers create tasks by invoking existing Java methods that will be executed on a device with minimal changes in the source code. Listing 1 shows a pseudocode of the task API. As shown, a group of tasks (t1 and t2) is created in the form of a task group (lines 6-8). Either the whole task group or individual tasks can be scheduled on the same, or different, heterogeneous device(s). Also, they can execute in parallel and all data dependencies (and data copying) is handled transparently by the VM with the help of the task scheduler and the runtime.

Tasks in our system encapsulate existing Java methods. In the example shown in Listing 1, the task t1 refers to the Compute bfs Java method while the task t2 refers to the Compute.mapReduce Java method. Those methods contain legal Java code that can include Java objects, exceptions, etc. The runtime system, in combination with the JIT compiler and the heterogeneous VM, compile and execute the existing Java code to the target accelerator via OpenCL.

```java
public class Compute {
  public static void bfs(int, out) { ... }
  public static void mapReduce(int, out) { ... }
}

public static void main(String[] args) {
  Schedule TaskGroup "gl" {
    task "t1" :: Compute::bfs, inA, outA
    task "t2" :: Compute::mapReduce, inB, outB
  }
}
```

Listing 1: Example of the Task Parallel API.

**Runtime System.** The runtime system performs type inference and obtains sizes and meta-data needed for compiling and optimizing tasks. It will also build a Data Flow Graph (DFG) to optimize data dependencies between tasks and generate new bytecodes for orchestrating their execution on the heterogeneous device.

**Heterogeneous VM.** This component will execute the new bytecode generated by the runtime system in a bytecode interpreter, resulting essentially in a "VM-in-a-VM" aiming at heterogeneous hardware virtualization. The main VM is the Java Virtual Machine (JVM) while the secondary is the VM that virtualizes the heterogeneous devices. Our runtime system generates, at runtime, new bytecodes to execute tasks on heterogeneous devices. These new bytecodes are interpreted in the heterogeneous VM that optimizes and orchestrates the execution across the accelerators. The heterogeneous VM will also manage memory between Java and the device while inspecting hardware features of the underlying device. This allows adaptability and transparency of applications when running on heterogeneous hardware.

Listing 2 shows an example of this new bytecode where two kernels (k1 and k2) are launched for execution on two devices (DEV0 and DEV1) in lines 5 and 9. These two tasks correspond to the generated bytecode of Listing 1. Before running the kernel, on-device memory is allocated and data is copied from the VM (output data are also copied back to the host’s memory). Allocating, copying memory, and running kernels can be synchronous or asynchronous operations, allowing more task-level parallelism. Just before running a kernel, the heterogeneous VM compiles the input Java methods (tasks) for a specific device and executes the resulting code.

**Heterogeneous JIT Compilation.** The JIT compiler and the VM work together for bringing performance and portability using iterative compilation. The compiler will optimize the input tasks for the selected device using its hardware information during runtime. It will also report to the VM profiling information to improve thread scheduling, bringing adaptability of high-level code to the target hardware.

```java
1 start // start the Heterogeneous VM
2 mem.alloc inA // memory alloc on the device
3 mem.alloc outA // memory alloc on the device
4 copy.in inA // data transfer Host->Device
5 runParallel t1 DEV0 // run task t1 on device 0
6 mem.alloc inB // memory alloc on the device
7 mem.alloc outB // memory alloc on the device
8 copy.in inB // data transfer Host->Device
9 runParallel t2 DEV1 // run task t2 on device 1
10 copy.out outA // data transfer Device->Host
11 copy.out outB // data transfer Device->Host
12 finish // finish the VM
```

Listing 2: Bytecode example of heterogeneous MRE.

**ACKNOWLEDGMENTS**

This work is partially supported by the EPSRC grants PAMELA EP/K008730/1 and AnyScale Apps EP/L000725/1, and the EU Horizon 2020 E2Data 780245.
REFERENCES