To expose, or not to expose, hardware heterogeneity to runtimes

Shoaib Akram
Ghent University, Belgium

ABSTRACT

Recent semiconductor scaling trends are steering hardware towards greater heterogeneity. Heterogeneous processors and memories have emerged to promise efficiency and scalability. Software must take advantage of emerging hardware heterogeneity. We ask the question, "what is the right software layer to abstract the complexity of heterogeneous hardware?"

Historically, the OS is the first choice to abstract new hardware features. Programmers, virtual machine developers, and language implementers benefit because they do not need to worry about hardware details. On the other hand, the upper layers of the software stack, especially the language runtimes, contain rich semantic information about user applications, unavailable to the OS. This information can be useful in managing hardware resources better. The drawback is that it requires software changes, making hardware vendors depend on runtime developers. This paper discusses two case studies that show exposing hardware details to the Java language runtime improves key evaluation metrics for popular Java applications. We further discuss implications for implementation complexity, programming model, and the necessary hardware and OS support.

ACM Reference Format:
Shoaib Akram. 2019. To expose, or not to expose, hardware heterogeneity to runtimes. In *Proceedings of Workshop on Modern Language Runtimes, Ecosystems, and VMs (MoreVMs'19).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Increasing hardware heterogeneity is inevitable. Modern hardware is becoming heterogeneous due to two device scaling trends. Transistor scaling under Dennard's rule that promised increasing transistor counts at a constant power density has stopped. The result is a greater emphasis on energy efficiency across the computing spectrum. Furthermore, Moore's law that enables shrinking transistor sizes is slowing down leading scientists to look for more scalable technologies. Heterogeneity in processors allows for a wide range of power versus performance tradeoffs. Heterogeneity in memory promises performance and scalability.

Heterogeneous multicores. Heterogeneous multicore (HM) processors, e.g., ARM big.LITTLE, combine big (high performing) and small (low energy) cores to enable energy efficiency. Big cores run program instructions out of program order to exploit instruction-level parallelism (ILP) and memory-level parallelism (MLP). In contrast, small cores run instructions in the program order. Application phases that do not expose ILP or MLP lose little performance by running on small cores. Hardware vendors such as Qualcomm are pushing heterogeneity beyond core types. The Qualcomm Snapdragon 805 processor consists of integrated CPU, GPU, DSP, and a range of specialized accelerators and co-processors. According to analysts, Intel plans to unveil integrated FPGAs in Sklylake-based Xeon processors soon.

Hybrid memories. Heterogeneity is not limited to processors alone. Scaling DRAM cells to smaller sizes is becoming a challenge. Prominent memory vendors have responded by introducing new memory technologies. Emerging memory technologies are non-volatile, and production systems are variants of phase change memory (PCM). Intel's Optane memory is byte-addressable, persistent, and faster than solid state disks. Unlike DRAM, PCM consumes no idle power. The main shortcomings of PCM are: (1) latencies are higher than DRAM, and (2) write endurance is low. Heterogeneous (hybrid) memory combines DRAM and PCM to offer scalable memory systems with good performance and low energy.

Exploiting hardware heterogeneity. How best we take advantage of heterogeneous hardware is an important question. Here, we focus on heterogeneous processors and memories. At one end of the spectrum are approaches to utilize hardware heterogeneity that do not involve software. In the software stack, OS is the first choice to abstract new hardware features. Hardware and OS approaches give hardware vendors independence from software companies like Microsoft and Oracle. On the other hand, modern language runtimes such as those for languages like C# and Java have rich semantic information. These runtimes interact closely with the application and know about application needs and behaviors.

Scheduling heterogeneous multicores. Let us consider thread scheduling on HMs. A hardware-only solution can use hardware performance counters to quantify the ILP and MLP in different threads. A hardware scheduler can then decide which threads to run on big versus small cores. A software-oblivious approach requires hardware support for context switches. Alternatively, the OS can use performance counter hardware to schedule threads on HMs. These disregard semantic information available in the runtime.

To see when semantic information is useful, consider a multi-threaded application written using a producer-consumer programming pattern. One set of threads produce information, which is processed (consumed) by another set of threads. Producers and consumers communicate via shared queues. Scheduling of the producer threads on big cores does not lead to the best performance if consumers cannot keep up. Conversely, running consumers on the big cores wastes energy if producers are not fast enough. Semantic knowledge can guide scheduling by communicating the relative rate of the progress of producers and consumer to the OS. The mutator and concurrent garbage collection threads in a managed runtime such as the Java Virtual Machine (JVM) manifest a similar pattern. Other examples are software pipelining and Hadoop MapReduce.

Managing hybrid memories. How best to manage hybrid memories is another concern. Let us focus on mitigating the low write endurance (wear-out) of PCM. Hardware-only solutions perform wear-leveling, i.e., spread writes out across the PCM capacity, or treat DRAM as a cache for PCM. However, these solutions can not eliminate writes by placing frequently written pages in DRAM. The

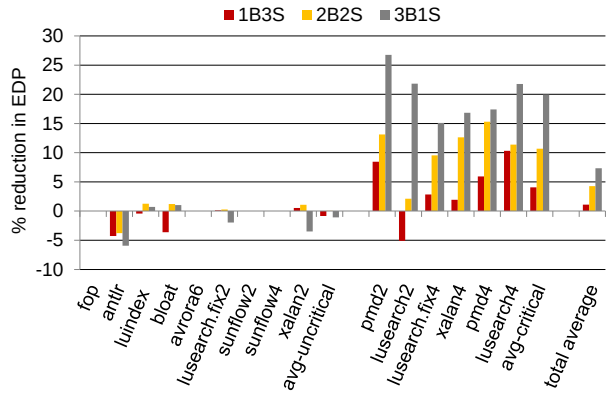

Figure 1: Percentage reduction in energy-delay product with GC-criticality-aware scheduling. 1B3S has one big core and three small cores. Multithreaded benchmarks have 2 and 4 appended to their names to show the number of mutator threads. *GC-criticality-aware scheduling improves energy efficiency over always keeping GC threads on small cores.*

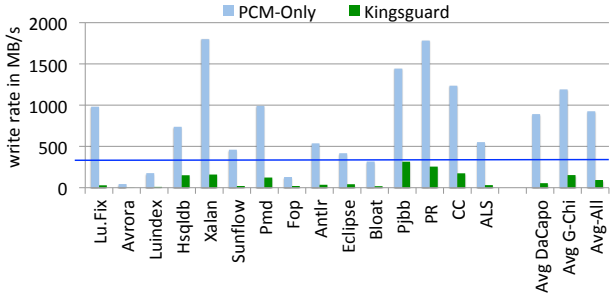

Figure 2: PCM write rates in MB/s. *PCM alone is impractical. Kingsguard collectors reduce PCM write rates significantly.*

OS, on the other hand, can identify and migrate frequently written pages in DRAM. Still, the OS approaches operate at a *coarse-grained* granularity and are reactive in nature.

Modern language runtimes such as the virtual machine implementations of Java, C#, and Python use garbage collection (GC) to manage virtual memory. GC has *fine-grained* knowledge about memory reference patterns. GC can thus migrate frequently written objects to DRAM to mitigate PCM wear-out.

The rest of the paper discusses two case studies that show exposing hardware details to the managed runtime results in better utilization of heterogeneous hardware.

2 RESULTS OF CASE STUDIES

We now present results from two case studies that support using semantic information in the runtime. We first discuss GC-criticality-aware scheduling on HMs [2]. The mutator and the concurrent GC threads in the Java runtime compete for big cores. Prior work recommends scheduling GC threads on small cores to save energy. We find that this approach makes GC critical for highly Java allocating applications. When GC becomes critical, it pauses the application to free unused heap memory. The result in overall performance degradation. However, scheduling of GC threads on the big cores takes big core cycles away from the application threads.

In GC-criticality-aware scheduling, the JVM determines and communicates the criticality of concurrently running GC threads to the OS. In turn, the OS boosts the priority of GC threads for the big cores.

Figure 1 shows the reduction in energy-delay product (EDP) with GC-criticality-aware scheduling over prior work. EDP is the product of energy and delay (execution time) of an application and captures the energy efficiency of a system. The benchmarks to the left are not GC-critical, and their allocation rates are low. On the right are GC critical applications. We observe that increasing the number of application threads increases GC criticality. On average, for a 3B1S system, using semantic information through GC-criticality-aware scheduling improves EDP by 8%.

GC-criticality-aware scheduling requires no hardware support or changes to the programming model. The OS needs to act upon critically signals which require minimal OS and JVM changes.

We now discuss write-rationing garbage collection for hybrid memories [1]. Write rationing garbage collection exploits object demographics and write patterns in Java applications to place frequently written objects in DRAM in a hybrid DRAM-PCM memory system. We discuss a specific collector here, namely Kingsguard. Kingsguard monitors writes on a per-object basis using barriers in the JVM. During a garbage collection, highly written objects are kept in DRAM and the rest in PCM.

Memory vendors place an upper limit on PCM write rate to guarantee a safe operation during the warranty period. We compute a recommended write rate of 140 MB/s for the Intel's Opteron memory from commercial data sheets. Figure 2 shows that a PCM-Only system is impractical as most applications write more than the recommended write rate. On the other hand, using Kingsguard collectors result in practical PCM write rates.

We also compare to a previously proposed OS solution. The results (not shown) show that the Kingsguard collector is more effective than the OS solution. More specifically, Kingsguard monitors fine-grained objects instead of coarse-grained pages and uses DRAM more efficiently.

Kingsguard requires no changes to the programming model. The implementation complexity of Kingsguard collectors is non-negligible but manageable. Kingsguard uses OS NUMA interfaces to interact with hybrid DRAM and PCM memories.

3 CONCLUSIONS

Modern hardware is increasingly becoming heterogeneous. An open question is what is the right software layer to abstract the complexity of heterogeneous hardware. This paper discusses the pros and cons of hardware, OS, and language runtime approaches to managing hardware heterogeneity. This paper demonstrates that semantic information in language runtimes can be instrumental in better management of heterogeneous hardware. It makes hardware vendors depend on language runtime developers, but requires only minimal OS support, and no changes to the hardware or the programming model.

REFERENCES

[1] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Write-Rationing Garbage Collection for Hybrid Memories. In *PLDI*.

[2] Shoaib Akram, Jennifer B. Sartor, Kenzo Van Craeynest, Wim Heirman, and Lieven Eeckhout. 2016. Boosting the Priority of Garbage: Scheduling Collection on Heterogeneous Multicore Processors. In *TACO*.