Hardware Performance Variation: A Comparative Study using Lightweight Kernels

Hannes Weisbach[1], Balazs Gerofi[3], Brian Kocoloski[2], Hermann Härtig[1], and Yutaka Ishikawa[3]

[1] Operating Systems Chair, TU Dresden
weisbach@os.inf.tu-dresden.de, haertig@os.inf.tu-dresden.de
[2] Washington University in St. Louis
brian.kocoloski@wustl.edu
[3] RIKEN Advanced Institute for Computational Science
bgerofi@riken.jp, yutaka.ishikawa@riken.jp

Abstract. Imbalance among components of large scale parallel simulations can adversely affect overall application performance. Software induced imbalance has been extensively studied in the past, however, there is a growing interest in characterizing and understanding another source of variability, the one induced by the hardware itself. This is particularly interesting with the growing diversity of hardware platforms deployed in high-performance computing (HPC) and the increasing complexity of computer architectures in general. Nevertheless, characterizing hardware performance variability is challenging as one needs to ensure a tightly controlled software environment.

In this paper, we propose to use lightweight operating system kernels to provide a high-precision characterization of various aspects of hardware performance variability. Towards this end, we have developed an extensible benchmarking framework and characterized multiple compute platforms (e.g., Intel x86, Cavium ARM64, Fujitsu SPARC64, IBM Power) running on top of lightweight kernel operating systems. Our initial findings show up to six orders of magnitude difference in relative variation among CPU cores across different platforms.

Keywords: Performance variation; Performance characterization; Lightweight kernels

1 Introduction

Since the end of Dennard scaling, performance improvement of supercomputing systems has primarily been driven by increasing parallelism. With no end in sight to this trend, it is projected that exascale systems will reach multi-hundred million-way of thread level parallelism [1], which by itself poses a crucial challenge in efficiently utilizing these platforms. Further complicating things, the majority of current large-scale parallel applications follow a lock-step execution model, where phases of computation and tight synchronization alternate and imbalance

across components can lead to significant performance degradation. Additionally, unpredictable performance also complicates tuning, as it becomes difficult to tell apart performance differences induced by platform variability from the result of the tuning effort.

Although performance variability is a well-studied problem in high-performance computing (HPC), for the most part variability has historically been induced by either operating system or application *software*. For example, it has been shown that interference from the system software (a.k.a., OS jitter or OS noise) can have an adverse impact on performance [2,3,4,5]. This has led to several efforts in lightweight operating systems [6,7,8] that reduce OS jitter, as well as work in parallel runtimes that attempt to balance load dynamically across processors at runtime [9,10]. However, exascale computing is driving a separate trend in *hardware* complexity and diversity that may further complicate the issue. With the increasing complexity of computer architecture and the growing diversity of hardware (HW) used in HPC systems, variability caused by the hardware itself [11] may become as problematic as software induced variability. Examples of causes for hardware induced variability include differences between SKUs of the same model due to process variation [12] during manufacturing, the impact of shared resources in multi/many-core systems such as shared caches and the on-chip network, or performance variability due to thermal effects [13].

While system software induced variability can be addressed by, for instance, lightweight operating system kernels [14,7,15,16], HW variability is a latent attribute of the system. As of today, there is little understanding of how the degree of hardware induced variability compares to that induced by software, and whether or not this difference varies across different architectures. One of the primary issues with precisely characterizing hardware performance variability is that measurements of hardware variability need to be made in such a fashion that eliminates software induced variability as much as possible, but making this differentiation is challenging on large scale HPC systems due to the presence of commodity operating system kernels. For example, a recent study investigated run-to-run variability on a large scale Intel Xeon Phi based system [11], but because of the Linux software environment, it is currently difficult to attribute all of the variability exclusively to the hardware platform.

In this paper, we provide a solution to this problem by designing a performance evaluation framework that leverages lightweight operating system kernels to eliminate software induced variability. With this technique we systematically characterize hardware performance variability across multiple HPC hardware architectures. We have developed an extensible benchmarking framework that stresses different HW components (e.g., integer units, FPUs, caches, etc.) and measures variability induced by these components. Given that variability is a key measure of how well an architecture will perform for large scale parallel workloads, our work is a key step towards understanding the capabilities of new and emerging architectures for HPC applications and to help HPC architects and programmers to better understand whether or not the magnitude of variability induced by the hardware is an issue for their intended workloads.

This paper focuses on per-core performance variation with limited memory usage, i.e., limiting working set sizes so that they fit into first level caches. The results provided here constitute our first steps towards a more comprehensive characterization of the HW performance variability phenomenon, including measurements that involve simultaneous usage of multiple cores/SMT threads, higher level caches, the memory subsystem, as well as comparison across multiple SKUs of particular CPU models. Specifically, this paper makes the following contributions:

- We propose a benchmarking framework for systematically characterizing different aspects of hardware performance variability running on top of lightweight kernel operating systems.
- Using the framework we provide a comprehensive set of measurements on per-core run-to-run hardware performance variability comparing Intel Xeon, Intel Xeon Phi, Cavium ThunderX (64 bit ARM), Fujitsu FX100 (SPARC-V9) and IBM BlueGene/Q (PowerISA) platforms.
- We use our performance evaluation framework to highlight a number interesting architectural differences. For example, we find that some workloads generate six orders of magnitude difference between variability on the FX100 and the Xeon Phi platforms. We also demonstrate that the fixed work quantum (FWQ) test [17], often used for OS jitter measurements is not a precise instrument for characterizing performance variability.

The rest of this paper is organized as follows. We begin with related work in Section 2. We provide background information on lightweight kernels and the architectures we investigated in Section 3. We describe our approach in Section 4 and provide measurements and performance analysis in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

Performance variability is an age-old problem in high-performance computing, with a plethora of research efforts over the past several decades detailing its detrimental impacts on tightly coupled BSP applications [18]. There are many diverse sources of variability, ranging from contention for cluster level resources such as interconnects [19] and power, to "interference" from operating system daemons [5,4], or intrinsic application properties that make it challenging to evenly balance data and workload *a priori* – for example, when application workload evolves and changes during runtime.

To mitigate these classes of variability, the HPC community has generally leveraged two strategies: (1) lightweight operating systems that reduce kernel interference by eliminating daemons and other unnecessary system services, and (2) parallel runtimes that provide mechanisms to respond to variability by, for example, balancing load [9,10,13], or by saving energy by throttling power [20,21] on the portions of the system less impacted by the particular source of variability.

Despite these efforts, there are indications that performance variability is poised to increase not only as a function of system software and algorithmic challenges, but also as a function of intrinsic hardware characteristics. With architectures continuing to trend towards thousand-way parallelism with heterogeneous cores and memory technologies, other architectural resources such as buses, interconnects, and caches are shared among a large set of processors that may simultaneously compete for them. While it is possible that parallel runtimes can address the resulting variability to some degree, recent research results indicate that today's runtimes are not particularly well suited to this type of hardware variability [22]. Thus, we believe there is a need for a performance evaluation framework that can precisely quantify the extent to which intrinsic hardware variability exists in an architecture.

As we mentioned earlier, multiple studies have investigated performance variation at the level of an entire distributed machine, however, none of them utilized lightweight kernels to clearly distinguish software and hardware sources [18,11]. It is also worth noting that the hardware community has been aware of some of these issues, for example, Borkar et. al showed the impact of voltage and temperature variations on circuit and microarchitecture [23].

3 Background

3.1 Lightweight Kernels

Lightweight kernels (LWKs) [16] tailored for HPC workloads date back to the early 1990s. These kernels ensure low operating system noise, excellent scalability and predictable application performance for large scale HPC simulations. Design principles of LWKs include simple memory management with pre-populated mappings covering physically contiguous memory, tickless non-preemptive (i.e., co-operative) process scheduling, and the elimination of OS daemon processes that could potentially interfere with applications [15]. One of the first LWKs that has been successfully deployed on a large scale supercomputer was Catamount [14], developed at Sandia National laboratories. IBM's BlueGene line of supercomputers have also been running an HPC-specific LWK called the Compute Node Kernel (CNK) [7]. While Catamount has been developed entirely from scratch, CNK borrows a significant amount of code from Linux so that it can better comply with standard Unix features. The most recent of Sandia National Laboratories' LWKs is Kitten [8], which distinguishes itself from their prior LWKs by providing a more complete Linux-compatible environment. There are also LWKs that start from Linux and modifications are done to meet HPC requirements. Cray's Extreme Scale Linux [24,25] and ZeptoOS [26] follow this path. The usual approach is to eliminate daemon processes, simplify the scheduler, and replace the memory management system. Linux' complex code base, however, can be prohibitive to entirely eliminate all undesired effects. In addition, it is also difficult to maintain Linux modifications with the rapidly evolving Linux source code.

Recently, with the advent of many-core CPUs, a new multi-kernel based approach has been proposed [27,28,29,6]. The basic idea of multi-kernels is to run Linux and an LWK side-by-side on different cores of the CPU and to provide OS services in collaboration between the two kernels. This enables the LWK cores to provide LWK scalability, but also to retain Linux compatibility.

As we will see in Section 4, from this study's perspective the most important aspect of multi-kernel systems is the LWK's jitterless execution environment, which enables us to perform HW performance variability measurements with high precision. Note that several of the aforementioned studies considering lightweight kernels have investigated the jitter induced by the Linux kernel and thus we intentionally do not include results from Linux measurements in this work.

3.2 Growing Architectural Diversity in HPC

Over the course of the past two decades, the majority of HPC systems have deployed clusters of homogeneous architectures based on the Intel/AMD x86 processor family [30], reflecting the overall dominance and ubiquity of x86 for heavy duty computational processing during this period. Architects and applications programmers have largely been successful at gleaning maximum performance from these processors by extensively tuning and optimizing key mathematical libraries, as well as leveraging low latency, high bandwidth interconnects to allow workloads to scale well with the number of machines. Based on the large body of effort in this space, a critical mass developed around the x86 ecosystem, which fueled further development and productivity for many generations of HPC systems.

However, the exascale era has brought a new set of problems, stemming from the end of Dennard scaling and increasing power and energy concerns, which are driving a shift away from solely commodity x86 servers towards a more diverse set of chip architectures and processors. On the one hand, to continue to provide increasing levels of parallelism, chip architectures have turned to heterogeneous resources. This can be seen with many-core processors, such as Intel Xeon Phi, now deployed on several large supercomputers [30]. Furthermore, the emergence of heterogeneous processors has created a need for other types of heterogeneous resources; for example, high bandwidth memory devices are provided alongside DDR4 on Intel Xeon Phi chips to provide the requisite bandwidth needed by the many cores.

At the same time, a renewed focus on power and energy efficiency has caused the HPC community to consider a wider set of more energy efficient processor architectures. Due to its widespread use in mobile devices where power efficiency has long been a key concern, ARM processors are seen as one candidate architecture, with several research efforts demonstrating energy efficiency benefits for HPC workloads [31,32], as well as indications that ARM chips are on a similar performance trajectory as x86 chips before they started to gain adoption in HPC systems in the early 2000s [33]. Other processors with RISC-based ISAs, such as SPARC's SPARC64 processors used in Fujitsu's K-computer [34], present potential energy-efficient options for HPC.

Table 1. Summary of architectures.

Platform/ Property	Intel Ivy Bridge	Intel KNL	Fujitsu FX100	Cavium ThunderX	IBM BG/Q
ISA	x86	x86	SPARC	ARM	PowerISA
Nr. of cores	8	64+4	32+2	48	16+2
Nr. of SMT threads	2	4	N/A	N/A	4
Clock frequency	2.6 GHz	1.4 GHz	2.2 GHz	2.0 GHz	1.6 GHz
L1d size	32kB	32kB	64kB	32kB	16kB
L1i size	32kB	32kB	64kB	78kB	16kB
L2 size	256kB	1MB x 34	24MB	16MB	32MB
L3 size	20480kB	N/A	N/A	N/A	N/A
On-chip network	?	2D mesh	?	?	Cross-bar
Process technology	22nm	14nm	20nm	28nm	45nm

Whether focusing on diversity in ISAs or heterogeneity of resources within a specific architecture, it is clear that the HPC community is facing a range of architectural diversity that has largely not existed for the past couple of decades. In this paper, we carefully examine some of the key architectural differences across a set of architectures, with a focus on the consistency of their performance characteristics. While others have performed performance comparisons across these architectures for HPC [33] and more general purpose workloads [31], we focus on the extent to which performance variability arises intrinsically from the architecture.

3.3 Architectures

While our framework is configurable to measure both core-specific as well as core-external resources, in this paper we present a detailed analysis of key workloads utilizing only core-local resources. In each of these architectures, this includes L1/L2 caches, as well as the arithmetic and floating point units of the core. We study these resources to understand how and if different processor architectures generate variability in different ways.

Table 1 summarizes the architectures used in our experiments. We went to great lengths to cover as many different architectures as we could, given the condition that we needed to deploy a lightweight kernel. We used two Intel platforms, Intel Xeon E5-2650 v2 (Ivy Bridge) [35] and Intel Xeon Phi Knight's Landing [36]. We also used Fujitsu's SPARC64 XIfx (FX100) [37], which is the next generation Fujitsu chip after the one deployed in the K Computer. ARM has been receiving a great deal of attention for its potential in the supercomputing space during the past couple of years. We used Cavium's ThunderX_CP [38] in this paper to characterize a processor implementing the ARM ISA. Finally, we also used the BlueGene/Q [39] platform from IBM.

Some of these platforms suite multi-kernels by design offering CPU cores separately for OS and application activities. The KNL is equipped with 4 OS

CPU cores, leaving 64 CPUs to the application, while the FX100 and BG/Q have 2 OS cores and provide 32 and 16 application cores, respectively. This is indicated by the plus sign in Table 1. Except FX100 and ThunderX, all platforms provide symmetric multithreading. The cache architecture also exhibit visible differences across platforms. For example, the KNL has 1MB of L2 cache on each tile (i.e., a pair of CPU cores), which makes the overall L2 size 34MBs. Except Intel's Ivy Bridge, all architectures provide only two levels of caches. We couldn't find publicly available information regarding the on-chip network for all architectures, we left a question mark for those.

4 Our Approach: Lightweight Kernels to Measure HW Performance Variability

To provide a high precision characterization of hardware performance variability we need to ensure that we have absolutely full control over the software environment in which measurements are performed. We assert that Linux is not an adequate environment for this purpose. The Linux kernel is designed with general purpose workloads in mind, where the primary goal is to ensure high utilization of the hardware by providing fairness among applications with respect to access to underlying resources.

4.1 Drawbacks of Linux

While Linux based operating systems are ubiquitous on supercomputing platforms today, the Linux kernel is not built for HPC, and many Linux kernel features have been identified as problematic for HPC workloads, ranging from variability in large page allocation and memory management [40], to untimely preemption by kernel threads and daemons [5], and to unexpected delivery of interrupts from devices [41]. Generally speaking, these issues arise from the Linux design philosophy, which is to highly optimize the common case code paths with "best effort" resource management policies that minimize average case performance but that sacrifice worst-case performance. This is in contrast to the policies used in lightweight kernels that attempt to converge the worst and average case behavior of the kernel so as to eliminate software induced variability.

While the behavior of the Linux kernel can be optimized to some degree for HPC workloads via administrative tools (e.g., `cgroups`, `hugeTLBfs`, IRQ affinities, etc.) and kernel command line options (e.g., the `isolcpus` and `nohz_full` arguments), the excessive number of knobs renders this process error prone and the complexity of the Linux kernel prohibits high-confidence verification even for a well-tuned environment.

4.2 IHK/McKernel and CNK

Because of these issues, we instead rely on lightweight operating system kernels introduced in Section 3. Specifically, we used the IHK/McKernel [42], [6]

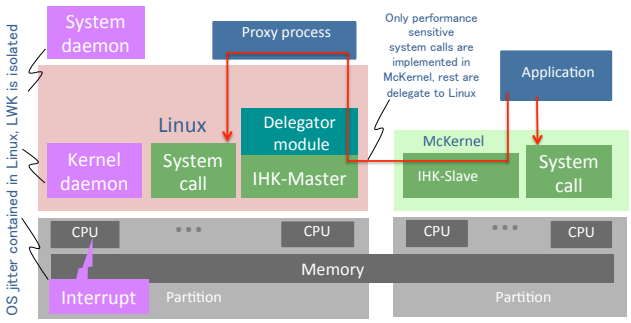

Fig. 1. Overview of the IHK/McKernel architecture.

lightweight multikernel in this study on all architectures except the BlueGene/Q where we took advantage of IBM's proprietary lightweight kernel [7]. While not the primary contribution of the paper, this work involved significant efforts related to porting IHK/McKernel to multiple platforms, in particular support for the ARM architecture.

The overall architecture of IHK/McKernel is shown in Figure 1. What makes McKernel suitable for this purpose is that we have full control over OS activities in the LWK. For example, there are no timer interrupts or IRQs from devices, there is no load balancing across CPUs and anonymous memory is mapped by large pages. All daemon processes, device driver and Linux kernel thread activities are restricted to the Linux cores. On the other hand, the multi-kernel structure of McKernel ensures that we can run standard Linux applications and it also makes multi-platform support considerably easier as we can rely on Linux for device drivers. As for BlueGene/Q, CNK provides a similarly controlled environment, although it is a standalone lightweight kernel that runs only on IBM's platform.

5 Performance Analysis

Previous studies on software induced performance variation relied on the FWQ and FTQ benchmarks to capture the influence of the system software stack on application codes. We hypothesize that simple benchmarks kernels like FWQ/FTQ or Selfish are insufficient to capture hardware performance variation. The full extent of hardware performance variation can only be observed when the resources which cause these variations are actually used. For basically empty loops which perform almost no computation this premise is not true. We propose a diverse set of benchmark kernels which exercise different functional units and resources as well as their combinations in an effort to reveal sources of hardware performance variation.

5.1 Benchmark Suite

Our benchmark suite currently consists of eight benchmark kernels and four sub-kernels. We selected our kernels from well-known algorithms such as DGEMM and SHA256, Mini-Apps, and micro benchmarks.

FWQ To test our hypothesis we have to include FWQ in our benchmark suite to provide a baseline. The FWQ benchmark loops for a pre-determined amount of times. The only computation is the comparison and increment of the loop counter.

DGEMM Matrix multiplication is a key operation used by many numerical algorithms. While special algorithms have been devised to compute a matrix product, we confine ourselves to naïve matrix multiplication to allow compilers to emit SIMD instructions, if possible. Thus, the DGEMM benchmark kernel is intended to measure hardware performance variation for double-precision floating point and vector operations.

SHA256 We use the SHA256 algorithm to exert integer execution units to determine if hardware performance variation measurably impacts integer processing.

HACCmk HACCmk from the CORAL benchmark suite is a compute-intensive kernel with regular memory accesses. It uses N-body techniques to approximate forces between neighboring particles. We adjusted the number of iterations for the inner loop to achieve shorter runtimes. We are not interested in absolute performance, but rather the difference of performance for repeated invocations.

HPCCG HPCCG, or High Performance Computing Conjugate Gradients, is a Mini-App aimed at exhibiting the performance properties of real-world physics codes working on unstructured grid problems. Our HPCCG code is based on Mantevo's HPCCG code. We removed any I/O code, notably printf() statements, and timing code so that only raw computation is performed by the kernel.

MiniFE MiniFE like HPPCG is a proxy application for unstructured implicit finite element codes from Mantevo's benchmark suite. We also removed or disabled code related to runtime measurement, output, and logfile generation so our measurement is not disturbed by I/O operations.

STREAM We include John McCalpin's STREAM benchmark to assess variability in the cache and memory subsystems. In addition we also provide the STREAM-Copy, STREAM-Scale, STREAM-Add, and STREAM-Triad as sub-kernels.

Capacity The Capacity benchmark is intended to measure the performance variation of cache misses themselves. The Capacity benchmarks does so by touching successive cache lines of a buffer that is twice the size of the cache to under measurement.

For most of the benchmarks the input parameters adjust the problem size and thus benchmark runtime. As discussed below, we decouple problem size and benchmark runtime so that we can adjust problem size and benchmark runtime independently. While our benchmarking framework allows to configure

benchmarks for arbitrary problem sizes, in this study we focus on problem sizes that fit into the L1 caches of our architectures. The idea is to eliminate or at least minimize the impact of the memory subsystem and shared resources beyond the L1 cache when we attempt to measure the performance variation of execution units. We adjust the working set to 90 % of the L1 data cache size, except for the Capacity benchmark, where we set the working set to twice the L1 data cache size.

We repeat a benchmark multiple times to fill a fixed amount of wallclock time with computation. A fixed time goal, in contrast to a fixed amount of work, allows us to dynamically adjust the amount of work to the performance of each platform and keep the total runtime of the benchmarks manageable. This is possible, because we are not interested in the absolute performance of each architecture but rather how performance varies between benchmark runs.

We select a benchmark runtime of 1 s to balance overall runtime and still have a long enough benchmark runtime to have meaningful results. After selecting the wallclock time, the benchmark suite performs a preparation run to estimate the number of times a benchmark has to be repeated to fill the requested amount of runtime with computation, which we call *rounds*.

We use architecture-specific high-resolution tick counters for performance measurement. For x86_64, we use the Time Stamp Counter with the `rdtscp` instruction. On AArch64 we use the `mrs` instruction to read the Virtual Timer Count register, CNTVCT_EL0, which is accessible from userspace. SPARC64 offers a TICK register, which we read with the `rd %%tick`-mnemonic. On the BlueGene/Q we use the `GetTimeBase()` inline function, which internally reads the Time Base register of the Power ISA v.2.06.

Timing measurements using architecture-specific high resolution timers are the lowest-level software-only measurements possible. We have considered employing performance counter data to narrow down sources of variability, but ultimately decided against it for the following reasons: (1) equivalent performance counters are not available on all architectures, (2) performance counters also vary between models of a single architecture, and (3) performance counter are occasionally poorly documented and/or do not work as documented. Nevertheless our framework has performance counter support for selected architectures, which we utilize to verify cache behavior. We plan to extend performance counter support to all architectures in the future.

Our benchmark suite is designed to run benchmarks on physical or SMT cores. Cores can be measured either in isolation by measuring core after core or a group of cores at once. The isolation mode is intended to measure core-local sources of variation, while the group-mode allows to measure variation caused by sharing resources between cores. Examples of interesting groups include all SMT-threads of a physical core, the first SMT-thread of all physical cores, or all SMT-threads of a processor. We restrict ourselves to measurements of all SMT-threads in isolation-mode in this first study of hardware performance variation. Note that during the measurement of a core in isolation-mode all other cores in the system are idle.

To obtain a measure of performance variation we repeat a benchmark 13 times and discard the first three iterations as warm-up. We use the remaining ten measurements of each SMT thread to determine the performance variation. We use two measures of variation in the study. The first measure normalizes the variation to the median performance of each core, the second to the minimum runtime measured for each core. We use the median-based measure when plotting performance variation for all cores of a machine. Given a vector $\mathbf{x}$, let $\widetilde{\mathbf{x}}$ be the median of $\mathbf{x}$. We visualize the variation by plotting the result of

$$(\mathbf{x} - \widetilde{\mathbf{x}})/\widetilde{\mathbf{x}} * 100.$$

Since this measure is based on the median variation might be positive as well as negative.

To reduce the variation of a single core into a single number, we calculate

$$\max \mathbf{x} / \min \mathbf{x} * 100 - 100$$

which yields the highest observed variation as percentage of the minimal observed runtime. Because the variations we observed between cores exhibited high fluctuation we decided against reducing the result to a single number, for example calculating a mean or average. Instead, we aim to preserve not only the minimal and maximal variation observed for each architecture, but also how the measured variations are distributed. Therefore, we present the measured variations in the form of a violin plot.

5.2 Results

We begin our evaluation by substantiating our claim that "empty loop benchmarks" such as FWQ are not suitable to measure hardware performance variation. In Figure 2 we plot the measured variation of each SMT core of our 2-socket x86_64 Intel Ivy Bridge E5-2650 v2 platform with FWQ and HPCCG. We set the working set size of HPCCG to 70 % of the L1 data cache size (32 KiB). We use the median-based variation , described in the previous paragraph, i.e. for each core we plot ten dots showing the percentage of variation from the median of each core.

The plot shows 30 of 32 SMT threads, because the two SMT threads of the first physical core run Linux, while the rest of the cores execute the benchmark under the McKernel lightweight kernel.

We turned the TurboBoost feature off, selected the performance governor, and set the frequency to the nominal frequency of 2.6 GHz. We additionally sampled the performance counters for L1 data cache and L1 instruction cache misses and confirmed that both benchmarks experience little to no misses.

Nevertheless all cores show significantly more variation under HPCCG than under FWQ. The difference cannot be accounted to cache misses, because even cores that show no data or instruction cache misses exhibit increased variation under HPCCG. In particular cores one to seven and 16 to 29 experience neither instruction cache nor data cache misses under HPCCG.

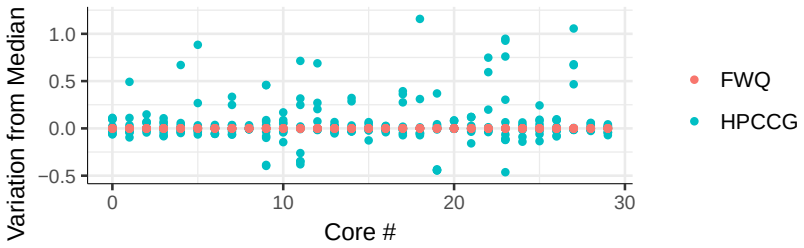

Fig. 2. Performance variation of FWQ and HPPCG on a dual-socket Intel E5-2650 v2.

After motivating the need for a diverse benchmark suite, we begin our comparison of performance variation. Because of the high dynamic range of performance variations within some architectures as well as across architectures we chose to plot the variation on a logarithmic scale. We keep the scale constant for all following plots to ease comparison between benchmarks. Lower values signify lower variation. Within a plot all violins are normalized to have the same area. The width of the violin marks how often different cores exhibited the same or at least a similar amount of variation. The height of the violins is a measure of how variation beween cores fluctuates; a tall violin indicates that some cores show little to no variation and other cores exhibit high variation. In contrast a small or flat violin is the result of cores having similar or even equal variation.

We treat CPUs as black boxes because CPU manufacturers and chip designers are not likely to share their intellectual property (i.e., chip designs and architectures), which are required to exactly pinpoint the sources of variability. We have considered using performance counters to narrow down sources of variability but dropped the idea due to the problems with performance counters iterated in the previous subsection.

First we present our results for the FWQ benchmark, plotted in Figure 3. The small violins in Figure 3 already indicate very low variation. A lot of measurements, particularly for the FX100 and BlueGene/Q systems, show no variation at all, i.e. we measured the same number of cycles. Because zero values become negative infinity on a logarithmic scale, we clipped the values at 0.5×10^{-7} % to avoid distortion of the plots caused by non-plottable data.

Nevertheless the plot clearly shows KNL with the highest variation of all platforms, while BlueGene/Q and FX100 show the lowest variation. To help the reader to put these variation measurements into perspective we note that the higher end of the ThunderX violin at 10^{-6} % corresponds to a "variation" of a single cycle.

Next we analyze the results of the STREAM benchmark in Figure 4. STREAM contains memory accesses as well as few arithmetic operations in its instruction mix. Although the working set is small enough to fit in the L1-cache we still see cache misses on architectures where we have support for performance counters. The observed variation increases for all architectures dramatically. The STREAM

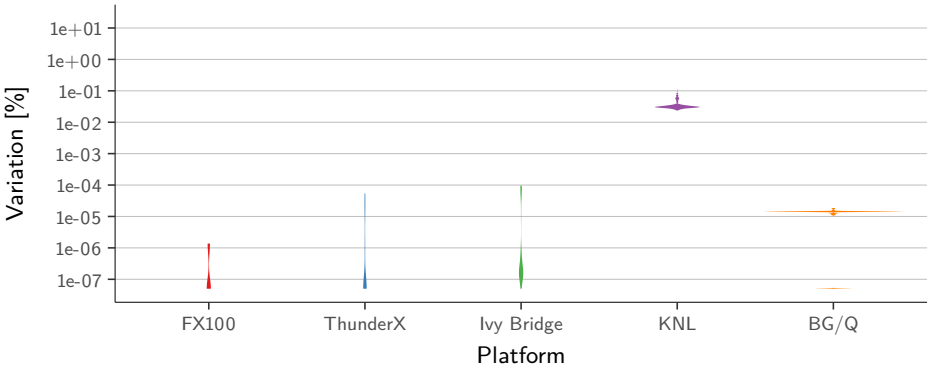

Fig. 3. Hardware performance variation under the FWQ benchmark.

benchmark seems to have the least impact on variation on the ThunderX platform, where the variation only increases by one order of magnitude.

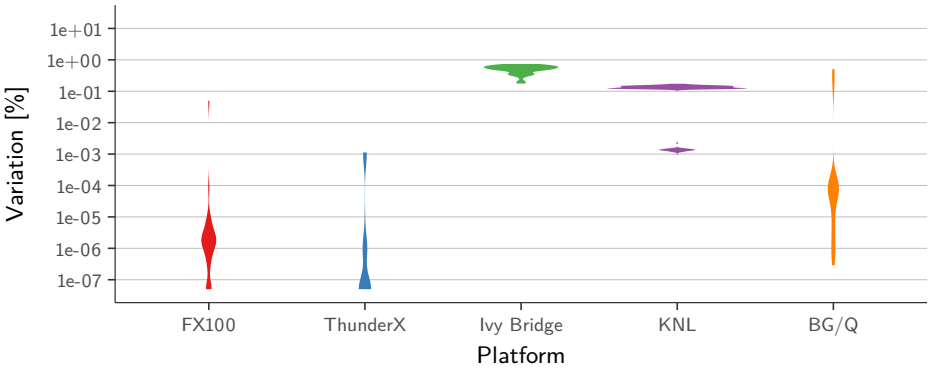

Fig. 4. Hardware performance variation under the STREAM benchmark.

The Capacity benchmark is similar to the STREAM benchmarks, but here the memory subsystem has to deal only with a single data stream. No computation is performed on the data, but the working set size is twice the size of the L1 data cache to intentionally and deterministically cause L1 cache misses. While the FX100 experiences little variation, the variation on the ThunderX platform increases substantially. The KNL platform shows very similar results for both the STREAM and Capacity benchmarks.

We found that the different architectures exhibited diverse behaviour for the SHA256 benchmark. Despite the same L1 cache size and associativity, we observed no L1 data misses on the ThunderX platform but approximately 150k misses on the Intel Ivy Bridge platform. We decided to include the results

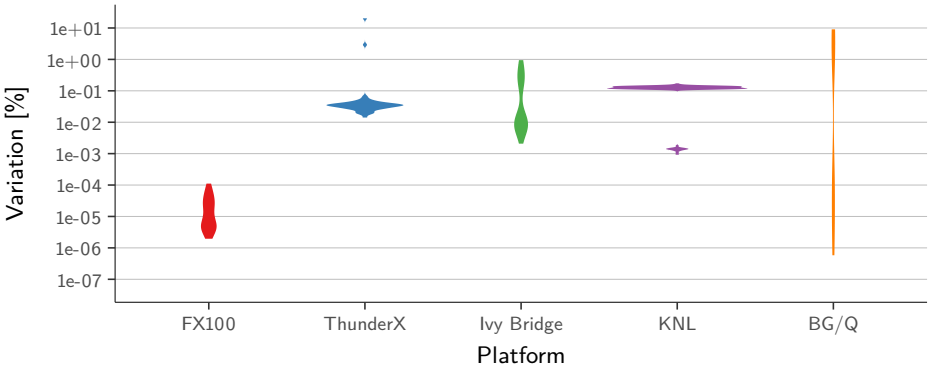

Fig. 5. Hardware performance variation under the Capacity benchmark.

as-is because we consider cache implementation details also micro-architecture-specific. Another reason is that the number of L1 misses on Ivy Bridge show little variation themselves. The wide base of the violins on FX100 and ThunderX already indicate that a lot of cores experience no variation at all, while Ivy Bridge performs significantly worse and KNL shows an order of magnitude more variation still.

We expected the BlueGene/Q to be among the lowest variation platforms but our measurements do not reflect that. At this point we can only speculate that the 16 KiB L1 data cache and the only 4-way set associativity of the L1 instruction cache have influence on the performance variation. We reduced the cache fill level to 80 % so that auxilary data such as stack variables have the same cache space in 32 KiB and 16 KiB caches, but we could not measure lower cache miss number of lower performance variation.

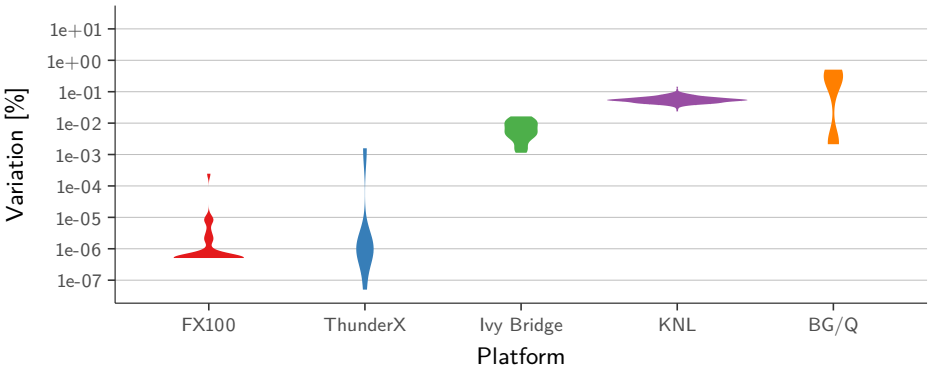

Fig. 6. Hardware performance variation under the SHA256 benchmark.

DGEMM is the first benchmark using floating point operations. This benchmark confirms the low variation of the FX100 and ThunderX platforms and the rather high variation of the Ivy Bridge, KNL and BlueGene/Q platforms. We saw high numbers of cache misses on the Ivy Bridge platforms and therefore reduced the cache pressure to 70 % fill level. We saw stable or even zero cache miss numbers for all cores of the Ivy Bridge platform, but variation did not improve.

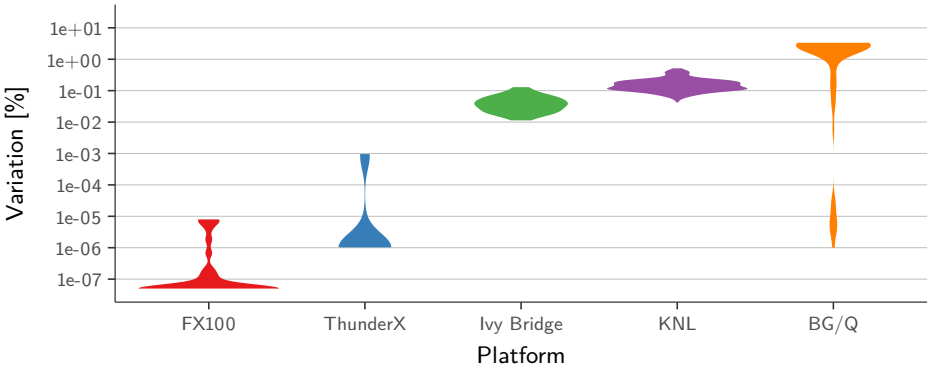

Fig. 7. Hardware performance variation under the DGEMM benchmark.

HACCmk has a call to the math library function `pow`, while Ivy Bridge and KNL instruction sets have `pow` vector instructions, we are not aware of such vector instruction on the FX100 and ThunderX platforms. FX100 and ThunderX show two oders of magnitude higher variation; 10^{-4} % corresponds to 100 cycles on the ThunderX platform. KNL and Ivy Bridge are more deterministic in the variation the exhibit, which results in "flatter" violins.

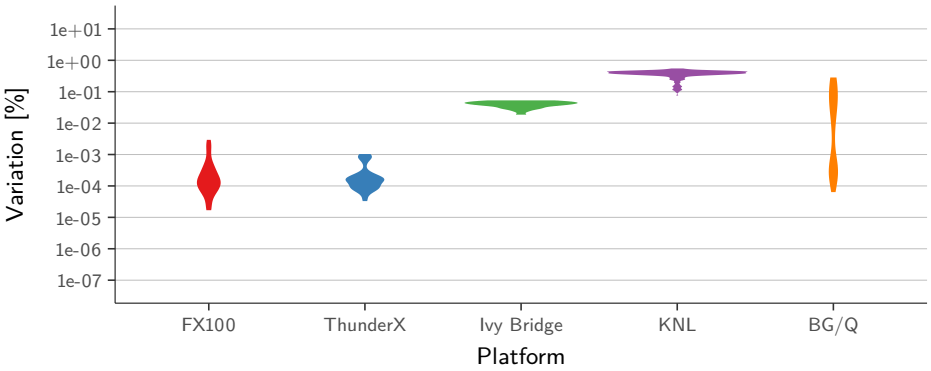

Fig. 8. Hardware performance variation under the HACCmk benchmark.

HPCCG is the only benchmark where the BlueGene/Q shows a variation close to our expectations. We also highlight that while the variation on the FX100 and ThunderX platforms show a reduction in their variation compared to DGEMM, Ivy Bridge and KNL show increased variation for this benchmark. We confirmed on both the Ivy Bridge and ThunderX platforms that no L1 data cache misses occur.

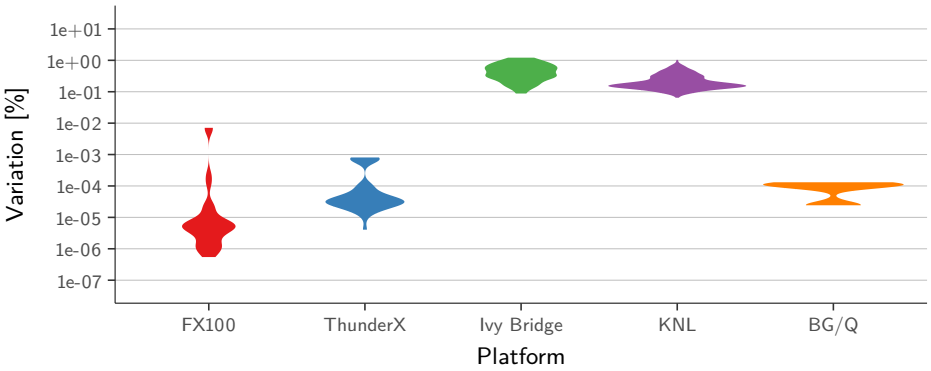

Fig. 9. Hardware performance variation under the HPCCG benchmark.

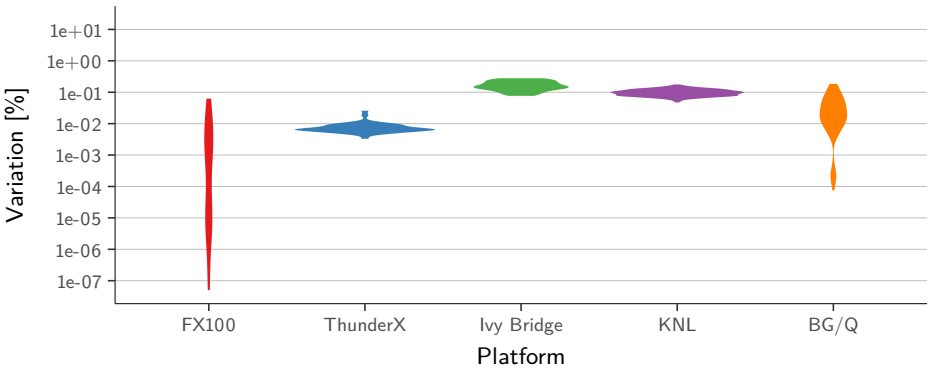

Fig. 10. Hardware performance variation under the MiniFE benchmark.

The MiniFE benchmark solves the same algorithmic problem as HPCCG. We expected similar results to HPCCG but our expectation was not confirmed by our measurements. The FX100 and ThunderX platforms show increased variation compared to HPCCG, while the Ivy Bridge and KNL platforms exhibit slightly lower variation.

6 Conclusion and Future Work

With the increasing complexity of computer architecture and the growing diversity of hardware used in HPC systems, variability caused by the hardware has been receiving a great deal of attention. In this paper, we have taken the first steps towards a high-precision, cross-platform characterization of hardware performance variability. To this end, we have developed an extensible benchmarking framework and characterized multiple compute platforms (e.g., Intel x86, Cavium ARM64, Fujitsu SPARC64, IBM Power). In order to provide a tightly controlled software environment we have proposed to utilize lightweight kernel operating systems for our measurements. To the best of our knowledge, this is the first study that clearly distinguishes performance variation of the hardware from its software induced counterparts. Our initial findings focusing on CPU core local resources show up to six orders of magnitude difference in relative variation among CPUs across different platforms.

In the future, we will continue extending our study focusing on higher levels of caches, the on-chip network, the memory subsystem, etc., with the goal of providing a complete characterization of the entire hardware platform.

Acknowledgments Part of this work has been funded by MEXT's program for the Development and Improvement of Next Generation Ultra High-Speed Computer System, under its Subsidies for Operating the Specific Advanced Large Research Facilities. The research and work presented in this paper has also been supported in part by the German priority program 1648 "Software for Exascale Computing" via the research project FFMK [43]. We acknowledge Kamil Iskra and William Scullin from Argone National Laboratories for their help with the BG/Q experiments. We would also like to thank our shepherd Saday Sadayappan for the useful feedbacks.

References

1. Markidis, S., Peng, I.B., Larsson Träff, J., Rougier, A., Bartsch, V., Machado, R., Rahn, M., Hart, A., Holmes, D., Bull, M., Laure, E. In: The EPiGRAM Project: Preparing Parallel Programming Models for Exascale. Springer International Publishing, Cham (2016) 56–68
2. Beckman, P., Iskra, K., Yoshii, K., Coghlan, S.: The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale. In: 2006 IEEE International Conference on Cluster Computing. (Sept 2006) 1–12
3. Ferreira, K.B., Bridges, P., Brightwell, R.: Characterizing Application Sensitivity to OS Interference Using Kernel-level Noise Injection. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. SC '08, Piscataway, NJ, USA, IEEE Press (2008) 19:1–19:12
4. Hoefler, T., Schneider, T., Lumsdaine, A.: Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. SC '10, Washington, DC, USA, IEEE Computer Society (2010) 1–11

5. Petrini, F., Kerbyson, D., Pakin, S.: The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In: Proceedings of the 15th Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Anaylsis. (SC '03) (2003)
6. Gerofi, B., Takagi, M., Hori, A., Nakamura, G., Shirasawa, T., Ishikawa, Y.: On the scalability, performance isolation and device driver transparency of the IHK/McKernel hybrid lightweight kernel. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). (May 2016) 1041–1050
7. Giampapa, M., Gooding, T., Inglett, T., Wisniewski, R.W.: Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. SC (2010)
8. Pedretti, K.T., Levenhagen, M., Ferreira, K., Brightwell, R., Kelly, S., Bridges, P., Hudson, T.: LDRD final report: A lightweight operating system for multi-core capability class supercomputers. Technical report SAND2010-6232, Sandia National Laboratories (September 2010)
9. Kale, L., Zheng, G.: Advanced Computational Infrastructures for Parallel and Distributed Applications. Wiley, Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects (2009)
10. Kaiser, H., Brodowicz, M., Sterling, T.: ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications. In: Proceedings of the International Conference on Parallel Processing Workshops. (ICPPW '09) (2009)
11. Chunduri, S., Harms, K., Parker, S., Morozov, V., Oshin, S., Cherukuri, N., Kumaran, K.: Run-to-run Variability on Xeon Phi Based Cray XC Systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '17, New York, NY, USA, ACM (2017) 52:1–52:13
12. Dighe, S., Vangal, S., Aseron, P., Kumar, S., Jacob, T., Bowman, K., Howard, J., Tschanz, J., Erraguntla, V., Borkar, N., De, V., Borkar, S.: Within-Die Variation-Aware Dynamic-Voltage-Frequency-Scaling With Optimal Core Allocation and Thread Hopping for the 80-Core TeraFLOPS Processor. IEEE Journal of Solid-State Circuits **46**(1) (2011) 184–193
13. Acun, B., Miller, P., Kale, L.V.: Variation Among Processors Under Turbo Boost in HPC Systems. In: Proceedings of the 2016 International Conference on Supercomputing. ICS '16, New York, NY, USA, ACM (2016) 6:1–6:12
14. Kelly, S.M., Brightwell, R.: Software architecture of the light weight kernel, Catamount. In: Cray User Group. (2005) 16–19
15. Riesen, R., Brightwell, R., Bridges, P.G., Hudson, T., Maccabe, A.B., Widener, P.M., Ferreira, K.: Designing and implementing lightweight kernels for capability computing. Concurrency and Computation: Practice and Experience **21**(6) (April 2009) 793–817
16. Riesen, R., Maccabe, A.B., Gerofi, B., Lombard, D.N., Lange, J.J., Pedretti, K., Ferreira, K., Lang, M., Keppel, P., Wisniewski, R.W., Brightwell, R., Inglett, T., Park, Y., Ishikawa, Y.: What is a lightweight kernel? In: Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers. ROSS, New York, NY, USA, ACM (2015)
17. : Fixed Time Quantum and Fixed Work Quantum Tests (Accessed: Dec, 2017). https://asc.llnl.gov/sequoia/benchmarks
18. Kramer, W.T.C., Ryan, C. In: Performance Variability of Highly Parallel Architectures. Springer Berlin Heidelberg, Berlin, Heidelberg (2003) 560–569

19. Bhatele, A., Mohror, K., Langer, S., Isaacs, K.: There Goes the Neighborhood: Performance Degradation due to Nearby Jobs. In: Proceedings of the 25th Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis. (SC '13) (2013)
20. Rountree, B., Lowenthal, D., de Supinski, B., Schulz, M., Freeh, V., Bletsch, T.: Adagio: Making DVS Practical for Complex HPC Applications. In: Proceedings of the 23rd ACM International Conference on Supercomputing. (ICS '09) (2009)
21. Venkatesh, A., Vishnu, A., Hamidouche, K., Tallent, N., Panda, D., Kerbyson, D., Hoisie, A.: A Case for Application-oblivious Energy-efficient MPI Runtime. In: Proceedings of the 27th Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis. (SC '15) (2015)
22. Ganguly, D., Lange, J.: The Effect of Asymmetric Performance on Asynchronous Task Based Runtimes. In: Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers. (ROSS '17) (2017)
23. Borkar, S., Karnik, T., Narendra, S., Tschanz, J., Keshavarzi, A., De, V.: Parameter Variations and Impact on Circuits and Microarchitecture. In: Proceedings of the 40th Annual Design Automation Conference. DAC '03, New York, NY, USA, ACM (2003) 338–342
24. Oral, S., Wang, F., Dillow, D.A., Miller, R., Shipman, G.M., Maxwell, D., Henseler, D., Becklehimer, J., Larkin, J.: Reducing application runtime variability on Jaguar XT5. In: Proceedings of CUG'10. (2010)
25. Pritchard, H., Roweth, D., Henseler, D., Cassella, P.: Leveraging the Cray Linux Environment core specialization feature to realize MPI asynchronous progress on Cray XE systems. In: Proceedings of Cray User Group. CUG (2012)
26. Yoshii, K., Iskra, K., Naik, H., Beckmanm, P., Broekema, P.C.: Characterizing the performance of big memory on Blue Gene Linux. In: Proceedings of the 2009 Intl. Conference on Parallel Processing Workshops. ICPPW, IEEE Computer Society (2009) 65–72
27. Wisniewski, R.W., Inglett, T., Keppel, P., Murty, R., Riesen, R.: mOS: An architecture for extreme-scale operating systems. In: Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers. ROSS, New York, NY, USA, ACM (2014)
28. Ouyang, J., Kocoloski, B., Lange, J.R., Pedretti, K.: Achieving performance isolation with lightweight co-kernels. In: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing. HPDC '15, New York, NY, USA, ACM (2015) 149–160
29. Lackorzynski, A., Weinhold, C., Härtig, H.: Decoupled: Low-Effort Noise-Free Execution on Commodity Systems. In: Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers. ROSS '16, New York, NY, USA, ACM (2016) 2:1–2:8
30. : Top500 supercomputer sites. https://www.top500.org/
31. Jarus, M., Varrette, S., Oleksiak, A., Bouvry, P.: Performance Evaluation and Energy Efficiency of High-Density HPC Platforms Based on Intel, AMD and ARM Processors. Springer Berlin Heidelberg (2013)
32. Rajovic, N., Rico, A., Puzovic, N., Adeniyi-Jones, C., Ramirez, A.: Tibidabo: Making the case for an ARM-based HPC system. Future Generation Computer Systems **36**(Supplement C) (2014) 322 – 334
33. Rajovic, N., Carpenter, P., Gelado, I., Puzovic, N., Ramirez, A., Valero, M.: Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC? In: Proceedings of the 2013 ACM/IEEE Conference on Supercomputing. SC (2013)

34. Miyazaki, H., Kusano, Y., Shinjou, N., Shoji, F., Yokokawa, M., Watanabe, T.: Overview of the K computer System. Scitech **48**(3) (2012) 255–265
35. Intel: Intel Xeon Processor E5-1600/E5-2600/E5-4600 v2 Product Families. https://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-1600-2600-vol-2-datasheet.html (2014)
36. Sodani, A.: Knights landing (KNL): 2nd Generation Intel Xeon Phi processor. In: 2015 IEEE Hot Chips 27 Symposium (HCS). (Aug 2015) 1–24
37. Yoshida, T., Hondou, M., Tabata, T., Kan, R., Kiyota, N., Kojima, H., Hosoe, K., Okano, H.: Sparc64 XIfx: Fujitsu's Next-Generation Processor for High-Performance Computing. IEEE Micro **35**(2) (Mar 2015) 6–14
38. Cavium: ThunderX_CP Family of Workload Optimized Compute Processors. (2014)
39. IBM: Design of the IBM Blue Gene/Q Compute chip. IBM Journal of Research and Development **57**(1/2) (Jan 2013) 1:1–1:13
40. Kocoloski, B., Lange, J.: HPMMAP: Lighweight Memory Management for Commodity Operating Systems. In: Proceedings of 28th IEEE International Parallel and Distributed Processing Symposium. (IPDPS '14) (2014)
41. Widener, P., Levy, S., Ferreira, K., Hoefler, T.: On Noise and the Performance Benefit of Nonblocking Collectives. International Journal of High Performance Computing Applications **30**(1) (2016) 121–133
42. Shimosawa, T., Gerofi, B., Takagi, M., Nakamura, G., Shirasawa, T., Saeki, Y., Shimizu, M., Hori, A., Ishikawa, Y.: Interface for Heterogeneous Kernels: A framework to enable hybrid OS designs targeting high performance computing on manycore architectures. In: 21th Intl. Conference on High Performance Computing. HiPC (December 2014)
43. : FFMK Website. https://ffmk.tudos.org