Operating System Process and Thread Migration in Heterogeneous Platforms

Robert Lyerly, Antonio Barbalace, Christopher Jelesnianski
Vincent Legout, Anthony Carno, Binoy Ravindran
Dept. of Electrical and Computer Engineering
Virginia Tech, Virginia, USA
{rlyerly, antoniob, bielsk1, vlegout, acarno, binoy}@vt.edu

ABSTRACT

Parallel and heterogeneous computing are here to stay. Moreover, diverse computational units are increasingly tighter-integrated in emerging heterogeneous platforms, sharing access to the memory bus. We argue that the traditional way we use heterogeneous platforms is obsolete. We propose new system software that enables programming these platforms as if they were an SMP, via the multi-threaded shared-memory programming model. Using this model provides better programmability, flexibility, and exploitability. We first provide an overview of the state transformation problem when migrating a thread between different-ISA processors. We then propose, build and evaluate a series of offline and runtime mechanisms that implement our design on top of Popcorn Linux ARM/x86. Results show that the overheads are minimal, thus proving the viability of our architecture.

1. INTRODUCTION

Heterogeneous computing platforms are now ubiquitous, powering systems from mobile devices to servers, and are here to stay [40]. Similar to parallel computing, heterogeneous computing strictly depends on software to fully exploit the hardware, complicating programming these systems. Hence, in this paper we advocate system software that improves the programmability of emerging heterogeneous platforms. Specifically, we propose OS and compiler mechanisms that enable the shared memory multi-threading programming model on such platforms, and thus process and thread migration among ISA-diverse processors.

Architectures Landscape. CPU/GPU setups are by far the most popular heterogeneous platform. However, new platforms deploying heterogeneous CPUs have begun to enter the market, e.g., ARM big.LITTLE [38], Intel Xeon and Xeon Phi [25], and x86 and ARM [31, 13]. In heterogeneous CPU/CPU platforms, each processor is general purpose and able to run an operating system (OS-capable [31]). Processors in heterogeneous computing platforms are becoming increasingly tightly-coupled [2, 38, 29, 3], i.e., they are interconnected through the memory bus, and therefore share memory which can be cache-coherent. Although platform-wide shared memory is a definite trend [20, 22], cache-coherency has been pinpointed as a fundamental issue hampering scalability to higher core counts [9, 15], opening the way to software programmed coherency [11, 33]. To exploit emerging heterogeneous platforms, applications should be able to use all available processor resources by seamlessly running across different processors to exploit diversity and by mapping data and optimizing sharing to reduce hardware overheads.

Heterogeneous Programming. The common practice in heterogeneous platforms is to start the application on the main processor (CPU) and during execution, offload parts of the application to a specific accelerator (e.g., GPU) that atomically executes a computation and returns the result. Offloading requires providing the remote processor with the code to be executed and the data on which it has to operate; in most architectures this requires a data copy. This computational model is acceptable for devices that provide limited control (non general-purpose), e.g., present-day GPUs. However, forcing the execution flow to return to the source processor at the end of each offloaded function is too rigid for emerging heterogeneous platforms [8]. In CPU/CPU platforms, the application can gain advantages by freely migrating between processors – execution migration on CPU/CPU platforms has been shown to enable better performance [8, 17] and lower power consumption [13, 42]

Contributions. Due to these trends, we argue that with increasingly heterogeneous and tighter-integrated platforms *offloading should be substituted by execution migration.* Hence, this paper introduces a system software architecture for process and thread migration in a multiple-kernel operating system (e.g., [7, 9]), atop OS-capable heterogeneous-ISA processors. Similarly to traditional SMP OSs, threads migrate among processors of different ISAs without any restriction. Therefore, we extend the multi-threaded shared memory programming model to heterogeneous platforms. We furthermore identify and evaluate the required system software mechanisms. This software design exploits a replicated-kernel OS which provides atomic operating system services [21] for thread migration. A prototype has been built atop Popcorn Linux on an ARM/x86 platform. We choose these two ISAs due to their popularity in today's server market and for their remarkable differences [13]. Note that this work does not address scheduling in these systems, but instead focuses on the mechanisms required for migration.

In Section 2 we survey the design space of the software for heterogeneous-ISA platforms, in Section 3 we describe the state transformation problem, and in Section 4 we introduce our system software architecture, which implements a specific solution in the design space. Section 5 discusses our implementation and Section 6 reports the costs of some of the proposed mechanisms. Section 7 concludes.

2. HETEROGENEOUS PLATFORMS

Heterogeneous platforms are historically built with multiple processors (e.g., a CPU and GPU), each of which has

its own memory. These platforms increasingly resemble distributed systems, and previous works [10, 27] proposed applying distributed systems principles to heterogeneous platforms. In particular, message passing has been advocated as a way to hide diversity and enable scalability. However, it is unclear how programmers will be able to efficiently utilize emerging platforms with such a software architecture.

Programming Models. Most application software for heterogeneous platforms uses one of two programming models for execution. *Message-Passing*, popular in distributed systems, uses hard-coded communication channels to share data between multiple processes, each of which may be executing on a different ISA. Data is transferred between processes at programmer-defined intervals and in pre-agreed formats. Additionally, the mapping of processes to processors is done only at application startup and does not change during execution. *Code Offloading* is widespread in single-node heterogeneous platforms. Similarly to message-passing, offloading requires the developer to decouple an application into multiple execution environments with hard-coded data transfer points and pre-agreed data formats. Offloaded execution runs to completion and cannot be interrupted or migrated. However, offloading differs from message-passing in that code is passed along with data to processors. Neither programming model provides the flexibility of shared-memory, which allows applications to migrate among available processors at any time according to a scheduler policy, thus enhancing execution flexibility.

System Software. Traditional system software for heterogeneous platforms moves the burden of handling diversity outside the OS while providing minimal system abstractions to communicate with remote processors (e.g., devices in Linux). Each set of different-ISA processors runs a distinct and independent operating system. Hence, recent research in system software proposes tightly coupling distinct execution environments [9, 23, 31, 6], even if they are not OS-capable [34, 36, 26].

The Way Ahead. On emerging heterogeneous platforms [25, 24, 1, 20, 22], software running on different processors may share access to global memory. This means execution environments on different processors are no longer restricted to message passing for low-level communication, allowing developers to exploit shared memory. These new architectures also allow more execution flexibility – message-passing and offloading restricts migration and communication to predefined locations, whereas a shared memory model allows transparent migration at any code instruction. In [8] we demonstrated the advantages of the shared memory programming model compared to offloading.

2.1 Data Sharing and Conversion

Processors implementing different ISAs not only have different instructions and register sets, but different native data storage formats, defined as the ISA's application binary interface (ABI). Even if different-ISA processors share memory, they must agree on a common data storage convention or data must be converted between formats upon execution migration (this requirement extends to executable code, although in our design the compiler is responsible for the conversion by generating machine code for each architecture). Thus, the question is: *how must applications be built and run in order to make processes and thread migratable among processors in a heterogeneous platform?*

The space of possible solutions includes two dimensions: data storage format and data sharing. In the data storage format dimension, data is either stored in a common format or data is dynamically converted between ABIs at runtime. In the data sharing dimension, threads either share all data (and therefore all program data), or no data is shared and replication is required, necessitating consistency via message-passing. In this design space there are three solutions often used today: shared everything using a common format (e.g., SMP multithreaded software), shared nothing using a custom format (e.g., the Internet), and shared nothing using a common format (e.g., PGAS models in a cluster).

Execution on heterogeneous processors requires data conversion. The knowledge for conversion can be either embedded in the code by the compiler (e.g., a common format and application layout), or retrieved at runtime through metadata lookup [37]. The former has less runtime overhead than the latter, but potentially limits architecture-specific optimization. A middle-ground solution can retain performance benefits while allowing execution flexibility.

2.2 Level of Abstraction

In a heterogeneous platform, data sharing and conversion can be implemented at any level of the software stack – *where should data sharing and conversion fit into the software stack to best enhance efficiency and programmability?* Two main approaches have been proposed in the literature for heterogeneous and homogeneous systems.

Application. As mentioned before, applications are traditionally the main users of platform heterogeneity (although several works have explored accelerator usage in kernel space [39]). Seminal work from Attardi *et al.* [4] advocated for user-space process migration among heterogeneous processors, which was implemented by Smith and Hutchinson in the TUI system [37]. The TUI system implements a shared nothing model with full state conversion when an application migrates between processors. More recently, DeVuyst *et al.* [17], Venkat and Tullsen [42], and Barbalace *et al.* [13] introduced application migration among heterogeneous ISA processors that share memory, enforcing a (partially) common address space for threads running on each ISA.

Kernel service. Operating system-level sharing of data structures amongst diverse processors has also been explored. Chapin *et al.* [16], K42 [44], Corey [14], and Bauman *et al.* [9] explored sharing on homogeneous ISA processors to attack scalability issues, while [23, 31, 10, 13, 28] extended this field to heterogeneous ISA processors. Barrelfish [9] proposed a shared nothing model amongst kernels, K2 [28] selectively implemented services in the shared nothing or shared everything design, and Popcorn Linux [6] implemented a strict shared nothing model. None of these works describe how to handle heterogeneity with completely disjoint ISAs. Kerrighed [30] proposed a shared mostly model based on distributed shared memory in cluster computing.

3. MODEL

In this section we discuss a model for multi-threaded applications running on a multiple-kernel OS. Kernels may run on different-ISA CPUs, with threads and processes migrating freely among these kernels. We consider an OS model in which the OS completely mediates IO, such that an application's address space exclusively maps main memory (i.e., no memory mapped devices). Kernel services provide appli-

cations with access to devices across kernels.

Application. The state of an application is a collection of variables and machine code. Each multi-threaded application includes per-thread state and per-process state. If the application is multi-process the model extends to sharing between multiple processes and includes a per-application state. The per-thread state includes thread local storage (TLS), user-space stack, and the user-space visible state of the CPU. The TLS data includes per-thread variables (e.g., GCC's declared `__thread` variables) as well as program libraries (e.g., `malloc` has per-thread variables to speed up memory allocation). The per-process state includes all other user-visible state that makes up the application's address space, such as all global data allocated in the heap or in the program's data sections and the application's machine code (i.e., the `.text` section).

Kernel Service. Each application also has a per-thread and per-process OS state. Moreover, the kernel has a per-CPU and a kernel-wide state. For an application thread executing in kernel-space, the per-thread OS state includes the stack, CPU registers, and per-thread OS data. The OS state of an interrupted application thread (e.g., signal) includes only the per-thread OS data. Note that in message-passing kernels, a thread's receiver buffer state belongs to either per-thread or per-process OS state, for both a thread executing in kernel-space or one that has been interrupted. The replicated-kernel OS design [7, 6] replicates OS services so that the per-application state of each service is the same on each kernel within kernels where the process is active. However, this design does not guarantee consistency of the kernel-wide system state among kernels. Instead, this depends on the service and the degree of sharing (i.e., for shared-everything, all kernels share a single OS state).

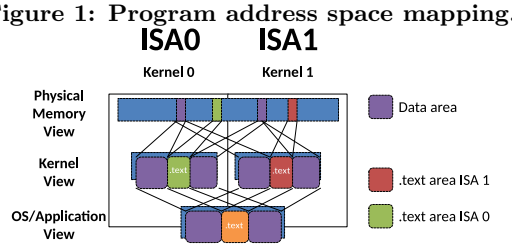
4. ARCHITECTURE

We propose a system software architecture for emerging heterogeneous-ISA CPU/CPU platforms that deploys one kernel per-ISA and uses distributed kernel services (a shared-nothing multiple kernel OS [8]). Application threads can migrate freely among kernels, and application threads on different kernels share memory. We enforce a common address space layout at compile or load time. For an application, each kernel configures the virtual memory system to map data at the same virtual addresses on all architectures. However, the `.text` section contains per-ISA code and is mapped per-architecture (shared-everything in userspace).

Previous approaches exploring execution migration [37] focused on migrating an entire process – we expand the scope to target thread migration. Hence we argue for a common per-process state, without which a distributed protocol (which maintains consistent views of the per-process state) and per-ISA format conversion routines must be deployed, which will likely add non-trivial execution overheads. Because per-thread state is exclusively accessed by the thread itself, the per-thread state need not be kept in a common format between ISAs. During migration, each member of the per-thread state can be converted to the destination format as needed (see Section 5). Moreover, a distributed shared memory (DSM) protocol will provide cache-coherent shared memory when not available in hardware (see Section 4.1).

Heterogeneous Continuations. This work extends Linux, a process model OS [21]. Each application thread has a user-space stack as well as a kernel-space stack. The proposed

Figure 1: Program address space mapping.



software architecture manages each stack differently. To facilitate process and thread migration, threads use the same user-space stack regardless of the ISA on which they are running. This design requires encoding the user-space stack in a common format or converting it during migration. Conversely, each thread has a per-ISA kernel-space stack. This is handled similarly to a continuation [19]. However, we associate a group of continuations with the same thread that runs on different-ISA CPUs (versus different threads).

Kernel threads do not migrate between kernels. However, application threads executing in kernel space can migrate under specific conditions. The replicated-kernel OS design distributes OS services, replicating global state among each kernel while maintaining local state per-kernel. Hence, an application thread that is executing code in kernel space cannot migrate during execution of a kernel service; otherwise, service atomicity is lost. This design does allow migration when the OS services are in a fully consistent state, such as just before or after an OS service (i.e., in the syscall layer).

4.1 Thread migration service and hDSM

A thread control block (TCB) exists on every kernel for a migrating application thread (`struct task_struct` in Linux). A task migration kernel service keeps TCBs consistent among kernels. For each ISA in the platform, the TCB refers to a different user-space machine code virtual memory area, kernel-space stack, and kernel- and user-space register state.

An application provides a per-ISA version of an application's machine code. When execution migrates between kernels, the machine code mappings are switched to those of the destination ISA. This is implemented by our heterogeneous distributed shared memory (hDSM) kernel service that aliases the `.text` section of each ISA at the same virtual address range. To provide the illusion of a single address space (despite the aliased regions), the hDSM service shares the same physical memory pages between kernels for cache-coherent systems, or implements a DSM protocol for non cache-coherent systems. Figure 1 depicts this mechanism, which is transparent to the application and portrays the exact same address space on each kernel.

Placing each symbol of the program at the same virtual address in each binary enables threads to transition from one ISA's binary to another. Alternatives exist, such as using dereferencing tables for each symbol; however, we believe this will increase the execution overhead for negligible gains in flexibility. This architecture considers migration only at equivalence points [43] (points at which an equivalent code continuation exists in another ISA's binary). Function boundaries are convenient migration points – the runtime system can decide if migration is worthwhile and explicitly call the kernel's migration service. The compiler should be modified to prepare such special binaries. To remove the

limitation of migrating only at equivalence points, we propose emulating the source ISA up until a migration point.

Kernel-space thread migration. Thread migration in kernel-space is only provided at points where the stack contains few frames and state transformation is trivial (e.g., the syscall layer or anywhere the scheduler is called). At these points, even if the thread in kernel-space is executing a service, the action will appear to be atomic by the application.

5. IMPLEMENTATION

In this section we discuss the mechanisms that we implemented in our prototype to provide a common state and state transformation for execution migration.

5.1 Offline Mechanisms

One kernel per ISA and distributed services. Operating system services should be written in order to be either distributed or centralized. In the latter case, a kernel that does not implement a service will send a request for the service to a remote kernel that implements it (i.e., a remote procedure call). In a replicated-kernel OS such as Popcorn Linux [8], services are mostly distributed, with a few being centralized. The server location is not fixed, although it is constant after being defined. A kernel should be compiled for, and run on, each ISA. The messaging layer and distributed services create the illusion of a single system [6].

Native compilation and common layout. An application that runs heterogeneously requires machine code for each ISA in the platform (similarly to multi-arch libraries implemented in Linux). A multi-architecture compiler toolchain produces a binary for each ISA in the platform, each of which contains a different `.text` section. Each program symbol has the same virtual address in every binary. This is achieved by enforcing a specific symbol order and alignment at linking time. Alternatively, symbol addresses can be resolved at load time (similarly to shared memory library loading).

Other requirements, such as data types having the same size and alignment (including processor word size), stack growth direction, and consistent endianness, are guaranteed by our prototype. Moreover, TLS can be laid out in a common format or transformed at runtime – we chose the former.

5.2 Runtime Mechanisms

Register Transformation. When an application's thread reaches a migration point, the operating system executes the migration service which transfers the thread's state between kernels. If the kernels are running on different ISAs, state conversion, including register contents, is required in order to resume execution on the destination ISA. However, migrating at function boundaries narrows down the number of registers to be converted, including the program counter, stack pointer, and frame base pointer.

Stack Transformation. If stack frames are not laid out according to a common ABI, the runtime must convert between frame layouts. In order to do the conversion, per-ISA metadata describing stack frame layout (i.e., locations of arguments and local variables) and the procedure to unwind frames from the stack is kept in the binary. The runtime repeatedly rewrites a single frame from the source ISA's stack to the destination ISA's stack, and can be applied eagerly to rewrite the entire stack (i.e., rewrite all activations on the stack) or rewrite on-demand if the cost must be amortized.

TLS Transformation. Similarly, if the TLS is not encoded in a common format (as we implemented) a runtime TLS transformation mechanism must be implemented. Compared to stack transformation, the TLS layout information is architecture-specific and encoded in the C library.

ISA Emulation. Emulation allows migration at any instruction in the code, therefore providing the same flexibility as thread migration in an SMP platform. Above, and in [6, 8], we propose to migrate exclusively at predefined application points. However, there can be situations in which more flexibility is needed to meet specific scheduling/mapping goals. Therefore, in order to migrate at any point in the code the runtime can emulate the source ISA until reaching a migration point, where state conversion can begin. Instruction emulation can be implemented by dynamic binary translation.

6. EVALUATION

We evaluated the cost of the mechanisms that implement transparent execution migration in heterogeneous platforms. All evaluations were run on a setup composed of an x86 64-bit machine and an ARM 64-bit machine interconnected through an IXH610 PCIe-to-PCIe non transparent bridge from Dolphin Interconnect Solutions [18]. The x86 CPU is an Intel i7-4790K with 4 cores (2-way hyperthreaded) at 4GHz, while the ARM CPU is an Applied Micro ARMv8 X-Gene with 8 cores at 1.6GHz. Both have 24GB of RAM. Microbenchmarks and the C version [35] of the applications from the NPB suite [5] were used in the evaluation.

6.1 Stack Transformation

Two factors determine stack transformation latency: stack depth and rewriting cost for individual frames. Stack depth, i.e., the number of frames on the stack, is directly proportional to cost – more frames must be re-written with a deeper stack. Rewriting costs consist of meta-data lookup (which varies depending on the number of arguments and local variables in a function) and re-writing (finding data in the source frame and copying it to the destination frame). Rewriting costs may be non-trivial due to deconstruction of complex data types (e.g., structs) and compiler optimization.

Figure 2 shows stack transformation latency for several microbenchmarks. **Empty**: functions have one argument and no local variables, leading to small lookup and re-writing costs. **Lookup**: functions have many arguments and local variables that are not used, causing heavy lookup costs but small re-writing costs. **Copy**: functions have many live arguments and local variables, leading to both heavy lookup and re-writing costs. The x-axis shows rewriting latency with varying stack depths. Note that NPB applications have an average stack depth of 3.71, with a worst-case of 6.

The high-powered x86 CPU outperforms the ARM CPU, rewriting the entire stack in under a millisecond for the majority of test cases. This means that the scheduler can migrate threads from x86 to ARM frequently without significant re-writing cost, increasing adaptability. The ARM CPU has a several-millisecond latency for re-writing larger stacks, meaning that migration from ARM to x86 should happen infrequently in order to minimize migration overheads. We expect this performance gap to shrink as the ARMv8 architecture evolves and compiler support becomes more mature (a 2.4GHz X-Gene1 reduces latencies by 30%).

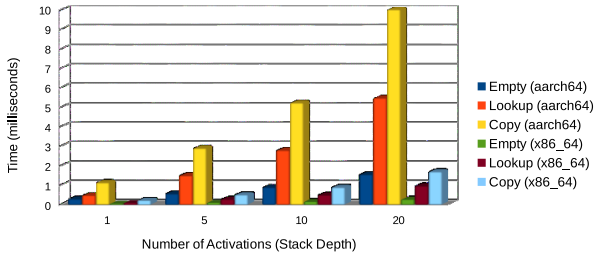


Figure 2: Stack transformation latency when eagerly re-writing stacks of varying depths.

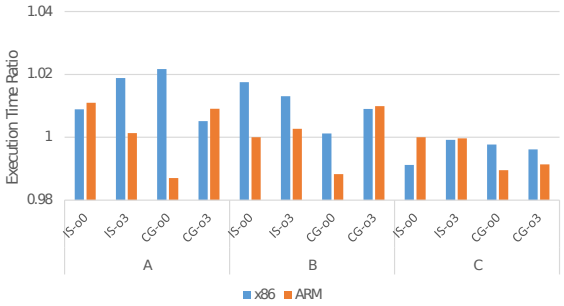


Figure 3: ARM and x86 execution time ratio compiling w and w/o alignment. NPB IS and CG, class A, B, and C. No optimizations (-O0) and with (-O3).

6.2 Common Layout

Imposing a strictly unified layout among binaries for different ISAs could potentially interfere with other optimizations when compiling a given application. To investigate this hypothesis, we compiled and ran a number of NPB applications with various problem sizes (classes A, B, and C) enabling different optimizations (GCC's "-O0", "-O3"). Figure 3 shows the execution time ratio of aligned vs. vanilla compilations on x86 and ARM, for the IS and CG benchmarks. Thus, a value greater than 1 indicates that an aligned compilation is slower. Each scenario is averaged over 100 repetitions. Enforcing the alignment, with or without optimizations, either reduces the execution time (up to 1.5%) or increases it (up to 2.1%). We noticed that long running benchmarks (class C) are less impacted if not improved by the alignment on both x86 and ARM.

We modified the gold linker [41] in order to emit the x86_64 TLS layout on ARM. Moreover, we updated the musl C library in order to use this TLS. Using a similar experiment as above, we observed that using the x86_64 TLS layout has a negligible impact on performance. With modified TLS, IS is 0.156%, 0.0238% and 0.00489% slower for class A, B, and C, respectively. Though the standard TLS layout is generally faster on ARM, the x86_64 layout has a small and diminishing impact as execution time increases.

6.3 ISA Emulation

We investigated the costs of using dynamic binary translation (DBT) for emulation with a prototype based on the QEMU machine emulator [12] (version 2.3). The prototype executes entirely on x86 (64-bit) in user-space and attaches to a running process, switching the process' execution from native to DBT (emulation). The prototype starts by pausing the application thread and pulling its register state. The register content is loaded into QEMU's internal CPU structures. It then loads the pages of memory allocated to the application into QEMU's internal memory map structures. At this point, QEMU begins translating and executing the binary. We stop translation and execution after a single basic block, though the prototype runs to completion.

We tested the prototype with the NPB IS class A benchmark. Our investigation showed that the primary costs of this process involve loading the pages allocated to the application into QEMU, 1200 μs, followed by the cost of starting QEMU, 400 μs. Translating the code from one ISA to another requires 60 μs and the costs of polling the registers and executing the translated code are negligible. However, porting this mechanism in kernel space will reduce such overhead from 1690 μs to roughly 490 μs.

6.4 Migration overhead

To evaluate the costs of thread migration in Popcorn Linux we migrated a thread from x86 to ARM, and back (thread ping-pong). This operation takes on average 9.2 ms, and consists of preparing for migration and migrating (1.2 ms), creating a remote execution context and executing an empty function on ARM, migrating back (7.7 ms in which execution is message-bounded), and resuming the execution on x86 (0.3 ms). The cost of register transformation is negligible. The inverse migration takes on average 8.7 ms. The thread ping-pong cost includes the cost of migrating memory pages needed by the remote architecture. The current hardware prototype does not have shared memory, therefore distributed shared memory is required. However, handling one page fault costs 140.9 μs on x86 and 80.2 μs on ARM. Note, that future hardware without platform-wide cache coherence will require a software page-ownership protocol with similar costs. However, these costs are biased by the hardware used in the prototype – transferring pages over PCI has a large impact on latencies, and having shared memory will substantially reduce costs [32].

7. CONCLUSION

Heterogeneous processors are increasingly populating computing systems at every scale, from mobile devices, to personal computers, to a rack, and the data center. Diverse processors may also co-exist in the data center as newer technology is integrated and brought online. Moreover, such diverse processors increasingly share the memory bus. These trends motivate us to reconsider process and thread migration in heterogeneous-ISA platforms.

We show that process and thread migration is feasible and we demonstrate that when exploiting shared memory, the cost of program state transformation can be kept low. On our prototype the mechanisms we employ introduce overheads of few milliseconds for stack transformation, common address space and TLS layout enforcement, runtime ISA emulation, and the OS migration service. With increasing program execution time, these overheads vanish. We believe that emerging hardware will further reduce these latencies.

8. ACKNOWLEDGMENTS

This work is supported by ONR under grant N00014-12-1-0880 and by AFOSR under grant FA9550-14-1-0163.

9. REFERENCES

[1] The Machine. http://www.hpl.hp.com/research/systems-research/themachine/, 2015.

[2] AMD. AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience. http://www.amd.com/us/Documents/48423_fusion_whitepaper_WEB.pdf.

[3] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden. Programming models for hybrid FPGA-CPU computational components: a missing link. *IEEE Micro*, 24(4):42–53, 2004.

[4] G. Attardi, I. Filotti, and J. Marks. Techniques for Dynamic Software Migration. In *In ESPRIT '88: Proceedings of the 5th Annual ESPRIT Conference*, pages 475–491. NorthHolland, 1988.

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks summary and preliminary results. In *Supercomputing '91*, 1991.

[6] A. Barbalace, A. Murray, R. Lyerly, and B. Ravindran. Towards Operating System Support for Heterogeneous-ISA Platforms. In *Proceedings of The 4th Workshop on Systems for Future Multicore Architectures*, 2014.

[7] A. Barbalace, B. Ravindran, and D. Katz. Popcorn: a replicated-kernel OS based on Linux. In *Ottawa Linux Symposium*, 2014.

[8] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran. Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 29:1–29:16. ACM, 2015.

[9] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44. ACM, 2009.

[10] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs. Your Computer is Already a Distributed System. Why Isn'T Your OS? In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, pages 12–12. USENIX Association, 2009.

[11] N. Beckmann and D. Sanchez. Jigsaw: Scalable Software-defined Caches. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 213–224. IEEE Press, 2013.

[12] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41. USENIX Association, 2005.

[13] S. K. Bhat, A. Saya, H. K. Rawat, A. Barbalace, and B. Ravindran. Harnessing Energy Efficiency of heterogeneous-ISA Platforms. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '15, pages 6–10. ACM, 2015.

[14] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57. USENIX Association, 2008.

[15] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8. USENIX Association, 2010.

[16] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-memory Multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 12–25. ACM, 1995.

[17] M. DeVuyst, A. Venkat, and D. M. Tullsen. Execution Migration in a heterogeneous-ISA Chip Multiprocessor. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 261–272. ACM, 2012.

[18] Dolphin Interconnect Solutions. Express IX. http://www.dolphinics.com/download/WHITEPAPERS/Dolphin_Express_IX_Peer_to_Peer_whitepaper.pdf.

[19] R. P. Draves. Control Transfer in Operating System Kernels. Technical Report MSR-TR-94-06, Microsoft Research, May 1994.

[20] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic. Beyond Processor-centric Operating Systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 17–17. USENIX Association, 2015.

[21] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and Execution Models in the Fluke Kernel. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 101–115. USENIX Association, 1999.

[22] S. Gerber, G. Zellweger, R. Achermann, K. Kourtis, T. Roscoe, and D. Milojicic. Not Your Parents' Physical Address Space. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 16–16. USENIX Association, 2015.

[23] G. C. Hunt and J. R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, Apr. 2007.

[24] T. Instruments. OMAP Processors. http://www.ti.com/lsds/ti/processors/dsp/media_processors/omap/products.page.

[25] Intel Corporation. Xeon Phi product family. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[26] S. Kim, S. Huh, Y. Hu, X. Zhang, E. Witchel, A. Wated, and M. Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 201–216.

USENIX Association, 2014.

[27] R. Knauerhase, R. Cledat, and J. Teller. For Extreme Parallelism, Your OS is Sooooo Last-millennium. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, HotPar'12, pages 3–3. USENIX Association, 2012.

[28] F. X. Lin, Z. Wang, and L. Zhong. K2: A Mobile Operating System for Heterogeneous Coherence Domains. *ACM Trans. Comput. Syst.*, 33(2):4:1–4:27, June 2015.

[29] MediaTek. MediaTek Helio x20. http://mediatek-helio.com/x20/.

[30] C. Morin, P. Gallard, R. Lottiaux, and G. Vallee. Towards an efficient single system image cluster operating system. In *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*, pages 370–377, Oct 2002.

[31] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 221–234. ACM, 2009.

[32] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out numa. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 3–18. ACM, 2014.

[33] N. Parris. Extended System Coherency - Part 1 - Cache Coherency Fundamentals. https://community.arm.com/groups/processors/blog/2013/12/03/extended-system-coherency--part-1--cache-coherency-fundamentals, 12 2013.

[34] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248. ACM, 2011.

[35] S. Seo, G. Jo, and J. Lee. Performance characterization of the nas parallel benchmarks in opencl. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 137–148, Nov 2011.

[36] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: Integrating a File System with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 485–498. ACM, 2013.

[37] P. Smith and N. C. Hutchinson. Heterogeneous Process Migration: The Tui System. Technical report, 1996.

[38] A. Stevens. Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7. Technical report, 2011.

[39] W. Sun, R. Ricci, and M. L. Curry. GPUstore: Harnessing GPU Computing for Storage Systems in the OS Kernel. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, pages 9:1–9:12. ACM, 2012.

[40] H. Sutter. Welcome to the Jungle. http://herbsutter.com/welcome-to-the-jungle/, 2012.

[41] I. L. Taylor. A New ELF Linker. In *Proceedings of the GCC Developers' Summit*, 2008.

[42] A. Venkat and D. M. Tullsen. Harnessing ISA Diversity: Design of a heterogeneous-ISA Chip Multiprocessor. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 121–132. IEEE Press, 2014.

[43] D. G. Von Bank, C. M. Shub, and R. W. Sebesta. A unified model of pointwise equivalence of procedural computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1842–1874, 1994.

[44] R. W. Wisniewski, D. da Silva, M. Auslander, O. Krieger, M. Ostrowski, and B. Rosenburg. K42: Lessons for the OS Community. *SIGOPS Oper. Syst. Rev.*, 42(1):5–12, Jan. 2008.