Operating System Support for Shared-ISA
Asymmetric Multi-core Architectures

Tong Li[†], Paul Brett[†], Barbara Hohlt[†], Rob Knauerhase[†], Sean D. McElderry[‡], and Scott Hahn[†]

[†]Intel Labs   [‡] Digital Enterprise Group
Intel Corporation
{tong.n.li,paul.brett,barbara.a.hohlt,rob.knauerhase,sean.d.mcelderry,scott.hahn}@intel.com

Abstract

Current trends in multi-core processor implementation scale by duplicating a single core design many times in a package; however, this approach can cause inefficient utilization of resources, such as die space and power. Recent research has proposed asymmetric cores as an alternative solution. This paper explores the design space for asymmetric multi-core architectures, and presents a case study and prototype of one design in which cores implement overlapping, but non-identical instruction sets.

We propose fault-and-migrate*, which enables the OS to manage hardware asymmetries transparently to applications. Our mechanism traps the fault when a core executes an unsupported instruction, migrates the faulting thread to a core that supports the instruction, and allows the OS to migrate it back when load balancing is necessary. We have also developed three approaches to emulate future asymmetric processors using current hardware. Preliminary evaluation shows that fault-and-migrate enables applications to execute transparently and incurs less than 4% overhead for a SPEC CPU2006[*] benchmark.*

1. Introduction

Advances in semiconductor technology have driven the hardware industry to integrate an increasing number of cores on-chip. Most existing multi-core processors consist of identical cores. However, as the number of cores continues to scale, this design can lead to inefficient utilization of chip real estate. Recent research [2, 3, 7, 8, 9, 15, 16, 22] proposes asymmetric (or heterogeneous) architectures as an alternative solution. For example, Kumar et al. [15] showed that, for the same die area, integrating out-of-order cores with a few in-order ones achieves comparable performance to traditional designs, but much higher energy efficiency.

There is a large design space for asymmetric architectures, ranging from cores differing only in clock speed to those differing in microarchitecture and ISA. The choices of hardware-software interface also vary. For example, hardware can hide asymmetry and expose the traditional view of identical cores. Alternatively, it can expose some cores as CPUs while others as coprocessors or peripherals. It can also expose all cores as CPUs and allow software to manage them completely.

This paper explores the design space for asymmetric multi-core architectures and presents a case study of one design, *instruction-based asymmetry*, in which cores have overlapping, but non-identical instruction sets. To enable transparent execution of applications, we propose an OS mechanism, *fault-and-migrate*, which traps the fault when a core executes an unsupported instruction and migrates the faulting thread to one that supports the instruction. Different policies exist as to when the thread can migrate back. For example, threads that execute unsupported instructions infrequently can migrate back quickly for load balance, whereas threads that frequently execute those instructions can stay longer on the new core to avoid the overhead of fault handling and migration.

We have implemented fault-and-migrate in the Linux kernel 2.6.20 and developed three approaches to emulate future asymmetric processors using current hardware. Our first approach emulates the removal of

1

floating-point (FP) instructions on a subset of cores. The second approach emulates the removal of an arbitrary subset of streaming SIMD extensions (SSE) instructions on any set of cores. In the third approach, we built a physically asymmetric, dual-socket system with two quad-core processors that have different SSE instruction sets. Preliminary evaluation using this system shows that fault-and-migrate enables transparent execution of applications and better load balancing. For a SPEC CPU2006* benchmark, it introduces less than 4% slowdown, compared to pinning the benchmark to a core with full support of its instructions.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 explores the design space for asymmetric architectures. Section 4 presents our case study, which discusses in detail the fault-and-migrate design, its Linux implementation, and our preliminary evaluation. We conclude with a list of hardware suggestions for improving OS management of asymmetric cores and discuss future work in Section 5.

2. Related Work

Heterogeneous processing traditionally has treated the "different" cores as coprocessors or peripherals. For example, NVIDIA's compute unified device architecture (CUDA*) exposes graphics processors as a coprocessor through libraries and OS drivers [19]. The Cell* processor runs applications on the Power Processor Element and offloads pre-defined code blocks to a set of Synergistic Processor Elements [8]. The EXOCHI system allows applications to offload compulation to a graphics processor via libraries and compiler extensions [22]. The C-CORE environment uses an Intel® IXP2400 network processor via the PCI interface to accelerate communication processing [17].

All of these designs require that the programmer partition the code into blocks and each block run on its specific type of core. For instruction-based asymmetric processors, this model burdens the programmer and can lead to inefficient core utilization, since a code block cannot utilize other types of cores. Our design allows the OS to manage all cores as traditional CPUs and dynamically handle instructions on cores that do not support them. Thus, we enable applications to transparently execute and fully utilize core resources.

The Multiple Instruction Stream Processing (MISP) architecture [9, 22] employs a proxy execution mechanism that is similar to fault-and-migrate. However, it requires hardware support for user-level faults and inter-processor communication, while our design has no such requirement and re-uses existing OS migration support. Furthermore, MISP runs OS services, such as system calls, and exception and interrupt handling, only on the OS-managed cores; any invocation of them on the application-managed cores triggers a fault and migration, whereas we incur this overhead only when a core executes a missing instruction.

Prior research also studied single-ISA performance-asymmetric architectures [2, 3, 6, 7, 15, 16, 18]. These studies are complementary to ours as we expect future processors to possess both performance and functional asymmetry.

3. Design Space

We explore the design space of asymmetric architectures in terms of types of asymmetry, hardware-software interface, and programming models.

3.1. Types of Asymmetry

We consider two types of asymmetry: performance and functional. With *performance asymmetry*, cores may differ in performance due to different clock speeds, cache sizes, and microarchitectures. All of these differences, however, manifest to software as only performance differences. Such designs enable higher energy efficiency while maintaining OS and application compatibility.

Alternatively, *functional asymmetry* allows cores to have different functionalities. For example, some cores may be general-purpose while others perform fixed functions such as encryption and decryption. Even if all cores are general-purpose, they can have different functionalities due to ISA differences. For example, to save die space, a processor may support SSE on some cores, but not others. We use the term ISA to refer to the portion of a system that is visible to software, including instructions, architectural registers, data types, addressing modes, memory architecture, exception and interrupt handling, and external I/O [20]. Without adequate support, applications compiled for one ISA can fail on cores with a different ISA,

even when the difference is small.

There are multiple dimensions of functional asymmetry, one for each aspect of the ISA. In the extreme case, a processor consists of cores with disjoint ISAs, such as Intel® IXP processors [21] and some implementations of integrated CPU and GPU cores. Alternatively, cores can have overlapping ISAs. The Cell* processor is an example where cores differ in most aspects of the ISA, but share the same data types and virtual memory architecture [8]. Due to the new instruction set, Cell* requires significant programming efforts for the software ecosystem, including the OS, compilers, libraries, tools, and so forth.

In this paper, we investigate a less radical design, *instruction-based asymmetry*, in which cores are identical in every aspect of the ISA, except that they can have overlapping, but non-identical instruction sets and architectural registers.

3.2. Hardware-Software Interface

We explore the design choices in terms of how the hardware exposes asymmetries and how software discovers them.

Exposing asymmetries. We consider three models for exposing hardware asymmetries:

- *Virtual-ISA model*: hardware (or firmware) hides asymmetries and presents to software an illusion of identical cores via a virtual ISA common to all cores [1]. This design greatly simplifies programming, but, on the other hand, significantly complicates hardware. Furthermore, lacking knowledge about application semantics hinders hardware from making resource management decisions that best meet application needs.

- *Coprocessor model*: hardware exposes a subset of cores as one or more coprocessors or peripheral devices. The OS code runs only on the main cores. Applications use the coprocessor cores as accelerators and access them via libraries and, in some cases, OS drivers. This is the model in Cell* [8], CUDA* [19], and EXOCHI [22]. Since it requires partitioning code to different types of cores, this model adds burden to the programmer and can lead to load imbalance, resulting in low system throughput.

- *CPU model*: hardware exposes all cores as CPUs, which share a physical address space, and the OS manages them directly. This model requires changes to core OS functions and both the processor and OS vendors to fully cooperate, leading to potentially longer time-to-market. However, with the OS managing all cores and handling asymmetry, this model enables applications to transparently execute and fully utilize the available cores in the system. Our case study assumes this model.

Discovering asymmetries. Except virtual-ISA, the other two models both require an interface for software to discover asymmetries. The coprocessor model can apply traditional I/O device discovery via memory-mapped I/O, special instructions, and so forth. Similarly, the CPU model can extend traditional CPU feature discovery interface, such as CPUID in Intel® Architecture (IA) [11], to both return the existing features of a core and indicate the missing ones available on other cores in the same processor. If hardware discovery is unavailable, the OS can build this topology, though less efficiently, by observing application faults and migrating applications between cores repeatedly until reaching one that supports the missing feature.

3.3. Programming Models

The choice of programming model is closely tied to how hardware exposes asymmetries. The virtual-ISA model allows the system to retain traditional multi-processor programming models, potentially providing maximum OS and application compatibility. The coprocessor model often adopts a master-slave programming style, where programs mostly run on the main cores and invoke libraries or OS services to offload pre-defined tasks to the coprocessor cores. This model requires that the programmer statically partition the code and map them to the appropriate types of cores.

With the CPU model, the modified OS dynamically schedules threads based on the instruction set they use and the load on each core, which simplifies application programming and enables better load balance. The programmer and compiler can use any instruction from the superset of the instruction sets of all cores. At run time, a missing instruction triggers a fault and the OS either resolves it or passes it to the application to al-

low the application to emulate the instruction or load a different binary.

4. OS Support for Instruction-based Asymmetry: A Case Study

This section presents a case study on the OS support for instruction-based asymmetric architectures.

4.1. Architecture

Our architecture makes the following choices:

- *Instruction-based asymmetry*: it allows manufacturers to re-use cores targeted at different market segments or from different generations. Compared to more complex designs, such as Cell[*], it greatly simplifies software enablement and backward compatibility.

- *CPU model*: it simplies both hardware design and application programming. With the OS managing all cores, applications can execute transparently and efficiently utilize core resources.

4.2. OS Support

We expect future asymmetric processors to exhibit both performance and functional asymmetry. Since functional asymmetry presents an immediate challenge to the execution of applications, this section focuses on the necessary OS support for it. Interested readers may refer to our previous studies [4, 18] for performance asymmetry.

Our design extends an existing OS with a fault-and-migrate mechanism. We assume that the hardware triggers a fault-type exception when executing an unsupported instruction. For IA cores, this exception already exists, known as the invalid opcode exception, or UD fault [12]. In the fault handler, our mechanism migrates the faulting thread to one of the cores that supports the faulting instruction. On the new core, the thread resumes execution and re-executes the faulting instruction.

To balance load, we allow the existing OS load balancer to later migrate the thread back to its original core. However, if the thread executes missing instructions frequently, it can thrash between cores. To balance the effort, we have studied two policies. First, we

allow a thread to migrate back after one quantum of execution on its new core. The second policy tracks instructions the thread retires on the new core that would otherwise fault on the old core. If none is found for an entire quantum, the thread can migrate back.

4.3. Discussion

Support for core pinning. Some OSes support core pinning, which confines a thread to only a set of cores. Thus, when deciding where to migrate a thread, we consider only cores that both support the faulting instruction and allow the thread to run on. To handle the case that no such core exists, our design includes a system call to allow applications to override fault-and-migrate. In this case, the OS sends a signal to the faulting thread, which can choose to either abort or invoke a handler of its own.

Migration versus emulation. Migration between cores with independent caches causes a thread to reload cache state with extra cache misses. Previous work [18] shows that this overhead is negligible on SMP systems, but can be significant on NUMA. We expect a similar trend in future multi-core systems. When migration overhead is high, our mechanism can choose to emulate a faulting instruction as opposed to migrate. If a thread faults frequently and the fault handling overhead is too high, binary translation could be used to avoid the unsupported instructions altogether [5, 10, 13].

Faulting in OS code. Certain OS code paths present specific issues to migration. In general, if the code is marked non-preemptible, it cannot be transparently migrated. Thus, we suggest that all non-preemptible OS code only use instructions supported by all cores. To prevent recursive faults, we make the same suggestion to code that implements fault-and-migrate.

4.4. Implementation

This section describes three prototypes for emulating functional asymmetry and our Linux implementation of fault-and-migrate.

4.4.1 Asymmetry Emulation

We have emulated an instruction-based asymmetric processor in three ways:

- *Disabling FP*: using an Intel® SMP system, this approach emulates the removal of the entire FP instruction set on a subset of cores by setting the EM, MP, and TS bits of register CR0 to 0, 1, and 1 [12]. Each FP instruction on these cores, including x87 and SIMD (i.e., MMX and all versions of SSE), triggers a device-not-available (NM) fault.

- *Disabling SSE*: this approach emulates the removal of selected set of SSE instructions on a subset of the cores. First, we disable FP instructions on these cores as described above. Next, we extend the NM fault handler in Linux with an x86 instruction decoder, which checks if the faulting instruction is one of those SSE instructions to be disabled on this core. If so, we migrate the faulting thread; otherwise, we allow the thread to continue on this core. For the latter, the fault handler first disables FP faulting such that the instruction that just faulted can successfully re-execute. Then, using one of the hardware breakpoint registers, the fault handler inserts a breakpoint at the instruction immediately following the faulting one. Upon reaching the breakpoint, the breakpoint exception handler re-enables FP faulting.

- *Asymmetric dual-processor (DP) platform*: We constructed a DP system with a quad-core Intel® Xeon® X5355 processor in one socket and a quad-core E5440 in the other. In addition to instruction-based asymmetry, this system also provides core frequency and L2 cache size asymmetry. The X5355 is 2.66 GHz with a 4 MB L2 cache and no support for SSE4.1, whereas the E5440 is 2.83 GHz with a 6 MB L2 and supports SSE4.1.

  Since asymmetric configurations are generally outside the specification of current processors, most existing BIOS disallows booting if it detects a mismatch in processor family, maximum frequency, voltage, or cache size. We modified the BIOS in our DP system to bypass these checks and allow the processors to operate potentially outside of their specified supply voltages.

4.4.2 Fault-and-migrate Implementation

We have implemented fault-and-migrate in Linux kernel 2.6.20. As shown with the three emulation prototypes, our design generalizes across platforms.

Fault handling. For the two FP-based prototypes, we modified Linux's NM fault handler to handle FP faults and statically configured a subset of cores to be de-featured. For the asymmetric DP platform, we modified the UD fault handler to handle faults of executing SSE4.1 on the X5355. To discover asymmetry, our code uses CPUID at boot time to construct a map of SSE4.1 capabilities for all cores. When a fault occurs, our fault handler changes the CPU mask of the faulting thread to include only cores capable of SSE4.1 and allowed by its original mask.

Migration. To migrate a faulting thread, the fault handler awakens Linux's migration thread on that core and suspends itself, which enables the handler to return quickly. When the migration thread runs, it migrates the faulting thread to an arbitrary core allowed by its new CPU mask. Alternatively, it can select the least loaded core. To minimize the cost of handling the fault, we currently choose the former approach and rely on Linux's periodic load balancing to handle any load imbalance that might occur.

Migrating back. We have implemented two policies to control when a thread is eligible to migrate back to its original core:

- *Always*: this policy always resets the thread's CPU mask to its original value after it completes one quantum on the new core.

- *Counter-based*: this policy counts the number of instructions the thread retires on the new core that would otherwise fault on the original. If zero for a quantum, it resets the thread's CPU mask.

Neither policies explicitly migrate the thread. By changing the CPU mask, we leverage Linux periodic load balancing to decide when and where to migrate the thread at a later time. The counter-based policy counts instructions differently for different emulation

approaches. For the one that disables FP, it leverages Linux's per-thread `fpu_counter` that counts the number of consecutive quanta in which the thread executes no FP instruction. For the other two, we use the observation framework in Knauerhase et al. [14], which tracks for each thread a set of hardware counter events. Since current Intel® hardware does not support counting arbitrary SSE instructions, nor SSE4.1 as a whole, we count SIMD instructions for each thread. This coarse-grained counting, however, can cause performance problems, as we show in Section 4.5.

Support for frequency asymmetry. Linux assumes a common core frequency and keeps global variables for the frequency (`cpu_khz`) and cycles per nanosecond (`cyc2ns_scale`). To support frequency asymmetry in our DP platform, we modified Linux to maintain these variables and their operations on a per-CPU basis.

4.5. Evaluation

This section discusses our preliminary evaluation. Since our DP system is physically asymmetric, while the other two prototypes are not, we present data from this system. As a first step, our evaluation has focused on how well fault-and-migrate supports instruction-based asymmetry. To help isolate it from the performance asymmetry aspects of our test system, we modified the BIOS to force all eight cores to operate at the same 2.66 GHz frequency. The resulting system still has cache size asymmetry, but presents a reasonable isolation for our evaluation.

Workload. Our workload consists of eight threads, one running `gamess`, a SPEC CPU2006[*] FP benchmark, and the remaining seven each running an infinite loop that keeps incrementing a counter. We compiled `gamess` using gcc 4.3 with `-msse4.1`, which enables the use of SSE4.1 instructions. The seven infinite loop threads contain no SSE4.1 instructions. We run these seven threads first, with each pinned to a different core but leaving one of the X5355 cores idle. Then, we run `gamess`. Since the X5355 core is idle, Linux automatically places `gamess` on it. Because the X5355 does not support SSE4.1, any SSE4.1 instruction invokes fault-and-migrate and
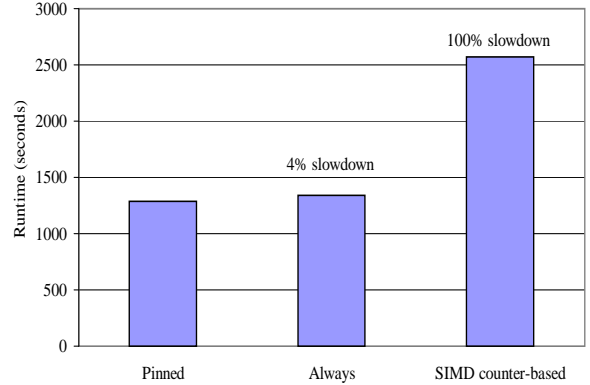

Figure 1: Fault-and-migrate performance.

causes `gamess` to migrate to one of the E5440 cores. The policies described in Section 4.4.2 allow `gamess` to restore its CPU mask at a later time. Since the original X5355 core becomes idle, Linux's load balancing allows `gamess` to migrate back to it. Thus, `gamess` migrates back and forth between two different types of cores, allowing us to stress-test fault-and-migrate.

Results. Figure 1 shows our performance results. The first bar is the runtime of `gamess` when it is pinned and the only thread on an SSE4.1-capable E5440 core, which we use as the baseline. The remaining two bars show its runtime under the above workload for the Always and SIMD counter-based policies. For Always, `gamess` experienced only 4% slowdown over the baseline, which includes both the overhead of fault-and-migrate and that due to a smaller L2 cache size (4 vs. 6 MB) on the X5355. The latter can account for a large fraction of this slowdown since `gamess` runs for the most time with a 4 MB cache in this setting, as opposed to 6 MB in the baseline. Thus, we estimate the overhead of fault-and-migrate to be much smaller than 4% and leave detailed measurements as future work.

One concern with the Always policy is that a thread may thrash between cores if it faults frequently, which motivated the counter-based policy. However, with the SIMD counter-based policy, `gamess` shows 100% slowdown, because it is SIMD-intensive—it retires at least one SIMD (e.g., SSE2) instruction per quantum. Thus, after its first migration, it stays on the E5440 core and never migrates back. Since there already exists an infinite loop thread on this core, `gamess` competes with it for CPU time, resulting in the halv-
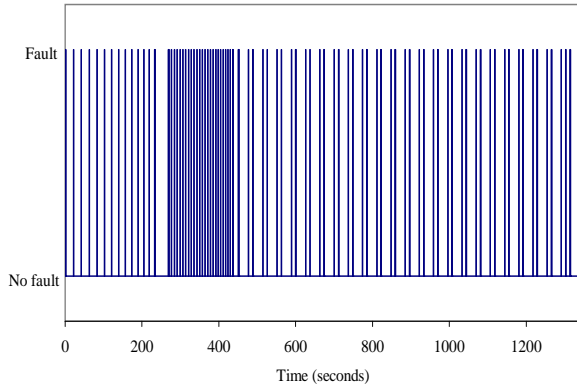
6

Figure 2: UD fault distribution of `gamess`.

ing of performance. This result suggests that, to obtain best performance, hardware should support fine-grained counting of asymmetric instructions.

Figure 2 shows the trace of UD faults that occurred during the 1342-second execution of `gamess` under the Always policy. Each vertical line represents one fault occurrence at that time. There are totally 120 faults, most of which are sparsely distributed, explaining why fault-and-migrate incurred low overhead and did not suffer from thrashing. The overhead can be higher when threads fault more frequently. Our future work will study how to efficiently handle these cases.

5. Conclusion

As multi-core processors continue to scale, the traditional approach of integrating identical cores can cause inefficient utilization of chip real estate. Asymmetric architectures provide an alternative cost-effective solution. This paper explores the design space of these emerging architectures and presents a case study of instruction-based asymmetric architectures in which cores have overlapping, but non-identical instruction sets. We proposed a fault-and-migrate OS mechanism that enables applications to execute transparently with low overhead. Using current hardware, we developed three prototypes to emulate future asymmetric processors. Our experience suggests three areas where hardware can help improve OS management:

- *Discovery of asymmetries*: hardware should improve existing CPUID or provide a new interface for the OS to efficiently discovery the hardware asymmetries.

- *Notification of missing features*: when a thread executes a missing instruction on a core, hardware should provide enough information to allow the OS to identify the specific instruction and determine where to migrate the thread.

- *Counting of missing instructions*: hardware should provide a counter for the retirement of each missing instruction on each core of the processor. This information can help the OS make more efficient scheduling decisions.

In our future work, we plan to evaluate our design with more workloads, investigate more sophiscated migration policies, and improve performance for applications with frequent faults.

Acknowledgments

References

[1] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A low-level virtual instruction set architecture. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 205–216, Dec. 2003.

[2] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's law through EPI throttling. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 298–309, June 2005.

[3] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 506–517, June 2005.

[4] J. M. Calandrino, D. Baumberger, T. Li, S. Hahn, and J. H. Anderson. Soft real-time scheduling on performance asymmetric multicore platforms. In *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2007.

[5] K. Ebcioğlu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 26–37, June 1997.

[6] A. Fedorova, D. Vengerov, and D. Doucette. Operating system scheduling on heterogeneous core systems. In *Proceedings of the First Workshop on Operating System Support for Heterogeneous Multicore Architectures*, Sept. 2007.

[7] S. Ghiasi, T. Keller, and F. Rawson. Scheduling for heterogeneous processors in server systems. In *Proceedings of the 2nd Conference on Computing Frontiers*, pages 199–210, May 2005.

[8] M. Gschwind. The Cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35(3):233–262, June 2007.

[9] R. A. Hankins, G. N. Chinya, J. D. Collins, P. H. Wang, R. Rakvic, H. Wang, and J. P. Shen. Multiple instruction stream processor. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 114–127, June 2006.

[10] S. Hu, I. Kim, M. H. Lipasti, and J. E. Smith. An approach for implementing efficient superscalar CISC processors. In *Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture*, pages 41–52, Feb. 2006.

[11] Intel. Intel® processor identification and the CPUID instruction. Application Note 485, Intel Corporation, Dec. 2007.

[12] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*. Intel Corporation, Feb. 2008.

[13] A. Klaiber. The technology behind Crusoe processors. White Paper, Transmeta Corporation, Jan. 2000.

[14] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multi-core systems. *IEEE Micro*, June 2008. To appear.

[15] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, Dec. 2003.

[16] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 64–75, June 2004.

[17] S. Kumar, A. Gavrilovska, K. Schwan, and S. Sundaragopalan. C-CORE: Using communication cores for high performance network services. In *Proceedings of the 4th IEEE International Symposium on Network Computing and Applications*, pages 171–178, July 2005.

[18] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, Nov. 2007.

[19] NVIDIA. *NVIDIA CUDA Programming Guide, Version 1.1*. NVIDIA Corporation, Nov. 2007.

[20] R. Ramanathan, R. Curry, S. Chennupaty, R. L. Cross, S. Kuo, and M. J. Buxton. Extending the world's most popular processor architecture. White Paper, Intel Corporation, 2007.

[21] M. Venkatachalam, P. Chandra, and R. Yavatkar. A highly flexible, distributed multiprocessor architecture for network processing. *Computer Networks*, 41(5):563–586, Apr. 2003.

[22] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. EXOCHI: Architecture and programming environment for a heterogeneous multi-core multi-threaded system. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 156–166, June 2007.