AM^3: Towards a hardware Unix accelerator for many-cores

Raphael Poss and Koen Koening

Abstract—This article advocates the use of new architectural features commonly found in many-cores to replace the machine model underlying Unix-like operating systems. We present a general Abstract Many-core Machine Model (AM^3), a proof-of-concept implementation and first evaluation results in the context of an emerging many-core, hardware multi-threaded architecture without support for interrupts. Our proposed approach makes it possible to reuse off-the-shelf multithreaded/multiprocess software on massively parallel architectures, without need to change code to use custom programming models like CUDA or OpenCL. Benefits include higher hardware utilization, higher performance and higher energy efficiency for workloads common to general-purpose platforms, such as in datacenters and Clouds. The benefits also include simpler software control over the hardware platform, an enabling factor for the further evolution of parallel programming languages.

Index Terms—Multi-cores, operating systems, computing models

1 Introduction

For better or for worse, the Unix operating system and its variants have long crystallized an abstract machine model that is now serving as foundation for most of the software industry: *processes* have one or more thread of execution sharing a heap and a stack in a virtual address space, and can perform *system calls* to an enclosing operating system (OS) that manages a set of processes with a shared set of virtual resources: file system, network, etc.

Even when programming languages define a different machine model, e.g. the STG for Haskell [1], users of applications written using these language are still exposed at run-time to the fact that the language's implementation simulates its own model on top of the Unix model: from the outside, the execution of a Haskell program is expressed on the underlying platform as threads ("capabilities" in Haskell) running in a process (the Haskell run-time system) [2], interacting with the enclosing OS via system calls; likewise, a Javascript program that uses the DOM environment as abstract model is ultimately expressed on the underlying platform as threads (one per browser tab in e.g. Firefox) running in a process (one per browser tab in e.g. Chrome, one for the entire browser in e.g. Firefox), that communicate with the enclosing OS via system calls to actually display elements on screen, receive user input or interact over the Internet.

This model is even largely reused in Clouds, albeit with different terminology ("virtual machine" vs. "process", "hypervisor" vs. "operating system"). The model remains largely unchanged in heterogeneous architectures, like those found in mobile phones: functions on the special hardware appear as foreign calls occurring within a thread/process running on a "host" processor under control of a Unix variant (e.g. Android).

A remarkable trait of the Unix abstract model is that ever since its advent in the 1970's, the *mechanisms* to implement the process abstraction in hardware have barely changed: memory addresses are translated in hardware automatically using a software-programmable translation unit; *the system call interface between processes and the OS is always implemented as a context switch within the same processor* and *time sharing is always implemented by connecting an external clock source to a hardware interrupt in each core*.

However, historically, interrupt handling was first implemented in hardware processors before Unix even existed in order to support manual interruption of long-running (or badly behaving) programs without shutting down the entire computer. The circuits in hardware that save the state of a task, restore the state of another task and switch control to it, *were only a necessary feature when the main computation tasks and "operator" tasks like shells or debuggers needed to share the same processor*. Unix then fortuitously piggy-backed on this hardware feature to implement time sharing and system calls, and this arrangement has persisted to this day. This begs the question: what if... What if cores were so cheap and so numerous that time sharing of single hardware threads wasn't a concern? What if cores were cheap and numerous because they were *small*, and the reason for that is that we could drop hardware support for interrupts and privileged code in all but a few of them? (1) Could we run Unix on that, i.e. reuse existing software as-is on the simple cores modulo recompiling the source code? (2) Then, what would be the benefits? (3)

These are the three main questions answered in this article. We first provide an answer to (1) by summarizing our perspective on the current architectural landscape in section 2 and describing precisely in section 3 why many-core chips would benefit from improved support in Unix-like systems. In section 4 we then describe a positive answer to (2) by defining a general machine model, AM^3, able to

• *R. Poss and K. Koening are with the Institute for Informatics, University of Amsterdam, the Netherlands.*
E-mail: see http://science.raphael.poss.name/

Manuscript received xxx; revised xxx.

support Unix on different many-core architectures. Section 5 then describes a proof-of-concept implementation on top of an emerging many-core architecture which was designed without support for interrupts. To answer (3), we start in section 5.4 by showing experimentally the benefits of our own implementation on common system tasks. We then compare our approach to related work in section 6. We discuss in section 7 the other indirect benefits that we have discovered and directions for future work, and conclude in section 8.

2 THE C/POSIX ABSTRACT MACHINE

In the decade 1990-2000, processor architectures have benefited from tremendous advances in manufacturing processes, enabling cheap performance increases from both increasing clock frequencies and decreasing gate size. These advances in turn enabled an explosive expansion of the software industry, with a large focus on uni-processors. This architecture model, that of the Von Neumann computer, had emerged at the end of the 1980's as the *de facto* target of all software developments.

Until the turn of the 21st century, system engineers could assume ever-increasing performance gains, by just substituting a processor by the next generation in new systems. Then they ran into two obstacles. The first was the *memory wall* [3]; to overcome this wall, processor architects have scrambled to preserve the uni-processor model for software by designing increasingly complex uni-processors using mainly branch predictors and out-of-order execution (OoOE) to automatically find parallelism in single threaded programs. Unfortunately, they eventually hit the *sequential performance wall* [4], [5], also known as "Pollack's rule" [6], i.e. the increasing divergence between single core performance and the power-area cost of the necessary hardware optimizations. To "cut the Gordian knot" [5], the processor industry has "given up" on single-core improvements alone and since shifted towards multiplying the number of processors on chip, now called *cores*.

Congruent with the advent of multi-cores, another "wall" is appearing: the increasing disparity between the chip size and the gate size causes *the latency between on-chip components (cores, caches and external interfaces) to increase relative to the pipeline cycle time*. This divergence is the on-chip equivalent of the memory wall: it causes mandatory waiting times in individual threads. Moreover, these latencies are becoming increasingly unpredictable, because of the larger software workloads and the increasing number of transient faults masked in hardware by automatically retrying operations.

The solution currently envisioned to overcome this wall is hardware multi-threading (HMT). HMT is a relatively old concept with always the same motivation: keep a processor busy while some thread(s) are waiting. From the simplest barrel designs [7], [8] to the fancier "hyper-threading" or simultaneous multi-threading (SMT) of recent Intel and Sun/Oracle products [9], [10], two features are shared by all HMT implementations. The first is what makes HMT relevant to tolerate on-chip latencies: *fast switching times*, made possible by provisioning separate physical program counters (PCs) and register files per hardware thread. The second was designed to make the adoption of HMT smoother in legacy software stacks: *full processor virtualization*, where each hardware thread also has its own identity with regard to address translation and its own interrupt routing logic so that it can be managed as a separate *virtual processor* (VP) by OS schedulers and trap handlers.

Overall, these successive developments were made with the assumption that the machine model observed by the OS is sacro-sanct: there may be more than one VP connected to the shared physical memory, but each VP must provide its own address translation unit and interrupt routing logic with a backward-compatible instruction set architecture (ISA). Yet this assumption is weakening. True, it is the case that this machine model is what all Unix-like OS kernels were originally designed for, and the software industry is dependent on the preservation of POSIX-like APIs that were originally defined for this model. However, the software industry has recently become accustomed to the idea of source-level software compatibility instead of a more costly cross-platform binary compatibility. This is made true by both the general adoption of open source infrastructure software and the diversification of ISAs forced upon an x86-dominated market by ARM and Oracle. In other words, we are entering an era where the preservation of *abstract models* and *functional interfaces* in source matters more than backward compatibility of binary code. But what do these abstract models look like?

For application processes, one needs not look further than ISO C 2011 [11] and POSIX [12]: the abstract machine provides one or more *threads* of execution that share a virtual address space; *system call interfaces* (also known as "syscall wrappers", e.g. `open`) provide access to system services, including starting, controlling and communicating with other processes and threads; *control flow* within single threads is decided by function calls, intra-function language flow control, inter-function jumps via `longjmp`/`ucontext` and inter-context jumps via `siglongjmp`, with extremely limited access to the hardware PC and registers; memory is abstracted via *virtual mappings* (e.g. `mmap`), used as back-end mechanism to implement heaps, stacks and memory-mapped I/O with little to no program control over address space layout; *signals* may be delivered to individual threads in reaction to asynchronous events, again via a standard interface (`sigaction`); and I/O is abstracted via numeric *descriptors* used as arguments to syscall wrappers; finally, *inter-thread synchronization* may be decided using a coherent shared memory, if available, and/or rely on OS-provided primitives such as mutexes, semaphores and message queues. All other programming languages in use on commodity hardware today are expressed within this abstract machine: as long as this model is preserved, our software stacks can be reused as-is modulo recompilation.

For an entire OS kernel, the essence of the machine model has been captured by virtual machine (VM) hypervisors, with Xen [13] as a poster child: as long as the machine provides VPs with a *privilege separation* between "user" and "system" code, a programmatic *interface to control address translation*, *configurable signals for scheduling*, and *external I/O via virtual packet-oriented networking interfaces or block-oriented storage interfaces*, it can run Linux or other Unix-like OS kernels and thus support any contemporary software stack.

Of remarkable interest in this article, the ability to deliver periodic signals in all VPs is still a general requirement, but *its demand is decreasing already*: the Linux and FreeBSD kernels, for example, can already operate a VP *tickless* [14], [15] when the VP runs only one process thread.

3 From multi- to many-cores: revisiting models

As the number of cores on chip grows, hardware architectures have started to diverge from the machine model currently presented to OS kernels in software. Are there features in the machine model which we could revisit in the light of recent architectural advances? In this section, we present three arguments: that hardware preemption is not needed in all cores, that increasing hardware complexity requires hardware acceleration for process/thread management, and that networks-on-chip (NoCs) are under-utilized by C/POSIX.

3.1 How necessary is hardware preemption really?

Generally, from the programmers's perspective, a thread runs sequentially, unininterrupted. Looking closely, not always.

There are four "consumers" of thread preemption in contemporary systems. The first is time sharing, to multiplex multiple threads or processes on a single VP. This consumer only exists as long as there are more logical threads/processes defined system-wide than there are VPs in the machine. The second is the collection of device drivers, which may need to receive asynchronous events from a larger number of different sources than there are VPs, however device drivers usually run on a small subset of all VPs available. The third is task reclamation, to suspend and/or remove a currently running task from a VP. For this use, we highlight here that reclamation is usually performed for entire processes at a time, i.e. upon all threads of a process simultaneously. The fourth is the intra-process abstract machine visible to application programmers, for in-application signal delivery. The C/POSIX model allows programs to configure signal delivery to arbitrary threads, but is this actually used in practice?

When examining the sources of open-source contemporary mobile, desktop or server software distributions (Android, GNU/Linux, FreeBSD), we can further narrow down which type of signals are actually used in applications, and more importantly *how*. The first observation is that relatively few programs actively control signal delivery. For those that do, they only control timer events (`SIGALRM`), process and channel control events (`SIGHUP`, `SIGINT`, etc.) and debugging (`SIGTRAP`). Perhaps surprisingly, although the C/POSIX standards have provisioned facilities to give applications control over the reaction to hardware faults (e.g. `SIGSEGV`, `SIGBUS`, `SIGILL`, `SIGFPE`), these facilities are only used in few "systems" programs (e.g. `valgrind`) and extremely rarely in application code. As to how signals are used, we have observed that only debuggers actively control fine-grained trap delivery to all threads in a process, if at all; in other code signal handling is configured to deliver all events to a "main" thread or a limited subset of the threads, dedicated to system I/O.

We summarize the situation as a starting assumption: *relatively few program threads in a modern Unix-like system require programmable preemption to be implemented by all VPs onto which they are mapped; the other threads merely need support for time sharing (and only when multiple threads are mapped to the same VP), and process-wide reclamation*.

3.2 Hardware heterogeneity and per-thread state

The concept of "memory" in a many-core chip has new dimensions that were not prevalent when C and POSIX were first designed. There are now "scratchpad" memories [16], [17] which are memory circuits accessible coherently from the VPs physically near to them, but either unaddressable from other cores or without cache coherency nor atomic semantics. The difference with register memory is that scratchpads can be indirectly addressed and may be shared by multiple threads. Configurable cache controllers are also becoming prevalent, where program code can specify per address range whether the memory is coherent with other VPs or not. Performance-sensitive programmers also increasingly demand visibility over the topology of the memory network and how logical objects in programs map to off-chip memory channels.

The reason why these reminders matter is that the *logical state* of a process or thread that must be maintained by the OS is becoming larger. It used to be only defined by the local VP state (PC, registers), open descriptors and virtual mappings. In the new context, the content of scratchpads, the state of local physical resources, optional custom cache parameters and whichever constraints a thread's code places on its physical placement in the system topology must be considered, too.

To summarize, as the number of cores grows, the size of the OS structures to manage individual processes or threads grow as well, and so does the complexity of saving or restoring the entire state of a process (e.g. to swap, to make space for another process, etc.). To keep a certain level of fluidity in OS schedulers, high-level process and thread management now need special optimizations, or, say, *acceleration*.

3.3 Are syscalls really "calls"?

In Unix terminology, the word "system call" is used when a user-level program requests a service from the OS. On most current architectures, the mechanism to trigger a syscall has little in common with the regular branching instructions used for regular calls; for one, the issuing program may not itself select the program counter reached after the branch. Instead, the word "call" refers to the historical requirement to suspend the invoking program while the OS service is running, when there is only one VP in the system, which in appareance makes the syscall behave like a subroutine call.

We see two main reasons to revisit this concept in the context of many-core processors. One is the growing demand for "asynchronous" syscalls (e.g. `aio_write` from POSIX.1) which reveals programmer acknowledgement of, and desire to exploit, the available concurrency between application code and OS operations. The second is the sheer cost of context switches required for privilege separation within one VP: registers and memory translation entries must be saved and reloaded for every switch from user code to system code and back again. When two or more VPs are available, the context of user code running on one VP needs not be saved, and the user code can keep running, while the system code runs on another VP. Interaction between the two can then happen using asynchronous messaging. This is a cornerstone of our proposal.

3.4 Efficient use of networks-on-chip

The physical reality of many-core chips is qualitatively different from traditional multi-processors: where historically processor(s) were connected to memory via *buses*, on-chip communication in modern multi- and many-core chips is now commonly *physically* supported by a high-speed packet-switched network [18], with routers between core tiles, or network-on-chip (NoC).

Yet the machine abstraction presented to OS kernels is one where the only way for VPs to communicate is via a shared memory (stores by one VP become eventually visible by loads from another VP) and a relatively clumsy inter-processor interrupt (IPI) delivery service for synchronization and periodic scheduling. This is the single machine abstraction currently used to implement in-kernel scheduler notifications (including thread creation and signal delivery), OS-based mutexes and message queues, stream-based (pipes & sockets) communication between threads, etc. It is also heavily used, with tens of interrupts delivered by second and per core on an idle system, up to several thousands per core and per second under I/O load or complex inter-thread synchronization patterns.

The effort required by chip architects to preserve this abstraction is staggering and very costly indeed. Globally shared memory between all cores requires increasingly complex cache coherency protocols, and causes overall core-to-core memory latencies to increase faster than their relative physical distance. For IPIs and TLB faults (virtual memory management), the programmatic interface of a late 20th century interrupt controller (the APIC) is physically emulated at each core on top of NoC messaging: an interrupt request by one core is translated locally to a data packet, carried over the network in packet form, then translated back by another emulation at the other end into an interrupt signal. Instead of emulating shared memory and interrupts, *why are we not using the hardware NoC directly in applications instead?*

The idea of programming a many-core like a distributed system on chip perhaps springs to mind here: from a bird's eye view perspective, cores with local scratchpads and connected with a packet-switched NoC do not conceptually differ from a network of workstations. There are however two reasons why a direct mapping of Unix-like networking on NoCs, as was already demonstrated in e.g. Intel's SCC [19] and Tilera's TILE, is neither efficient nor desirable. The first is that Unix-like networking requires either data copying between userspace and the network hardware, or physically shared buffers between the network interface and the processor, which are usually not available in NoCs. The second is that network operations require the overhead of (at least) one context switch on both sides, which in turn involves memory where process state is stored. In short, *Unix networking requires memory*. However, since memory is already shared anyways between VPs, using that directly is necessarily more simple and efficient than adding the overhead of a fully fledged network stack.

Meanwhile, other traditional uses of shared memory would greatly benefit from tighter NoC integration: the high-bandwidth, fine-granularity inter-process communication (IPC) facilities offered by most Unix kernels. Message queues, pipes and Unix domain sockets in particular are nowadays prevalent in any sizeable software system and support the majority of I/O operations in networked code. These services are furthermore fully abstracted by the OS which makes a direct mapping to NoC messaging possible without changes to application code.

In order to show that NoCs can be successfully leveraged in applications without the overhead of emulating a cache coherency protocol and interrupt delivery, three conditions must be met. The first is that neither inter-core cache coherency nor interrupts must be involved in the communication. The second is that the resulting improvements must be sufficient to justify a change in the model. The third is that existing code must benefit from the improvement without changes. Our proposal, which follows, meets all three.

4 AM^3: AN ABSTRACT MANY-CORE MACHINE MODEL

We keep most aspects of the traditional machine model. VPs are connected to a shared memory. Each VP has its own PC and local state distinct from other VPs. All memory operations undergo address translation; a *privilege separation mechanism* must exist with as minimal requirement that it must 1) prevent "user" code from altering its own address translation tables (isolation) and 2) guarantee that the behavior of "user" tasks do not prevent progress of "system" tasks (fairness). Where AM^3 differs from the traditional model is how this separation mechanism is implemented.

In the model inherited from uni-processor multi-programming, the configuration of privilege separation is performed via privileged instructions; the *current privilege level* stored in a status register determines whether privileged instructions are allowed or not (system mode vs. user mode); and a *context switch protocol* allows calls from user mode to system mode but with limited control by the user code over which code gets executed in system mode. In the ISA, context switching protocols commonly reuse the opcode for software interrupts, although Alpha, MIPS, ARM, x86-64 and some others also have a dedicated `syscall` instruction. In all cases, the circuits for context switches in hardware are shared with interrupt handling, which often has ramifications throughout the core micro-architecture: the pipeline must be flushed, outstanding instructions (e.g. pending memory operations) must complete, the VP's local state (PC, registers, status words) saved, then replaced by those of the new context, before execution resumes. In ARM and SPARC, a context switch slides the register window to a dedicated register file, however in Unix-like OS kernel this is merely used as trampoline and the handler running in the dedicated file eventually saves/restores a target context in the regular register file from memory. Our proposed model AM^3 breaks clean of these requirements.

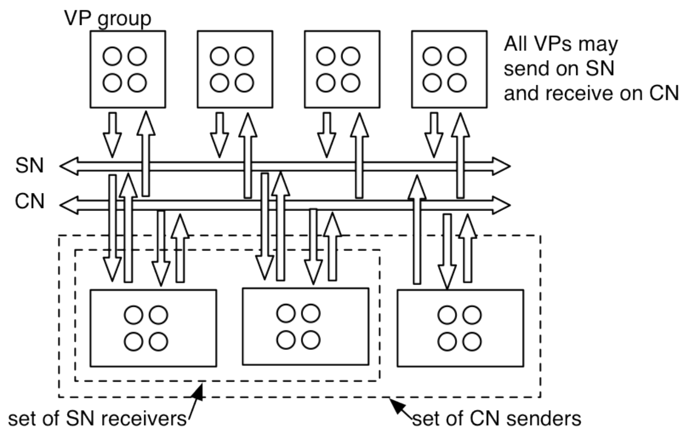

Figure 1. Proposed SN/CN system model.

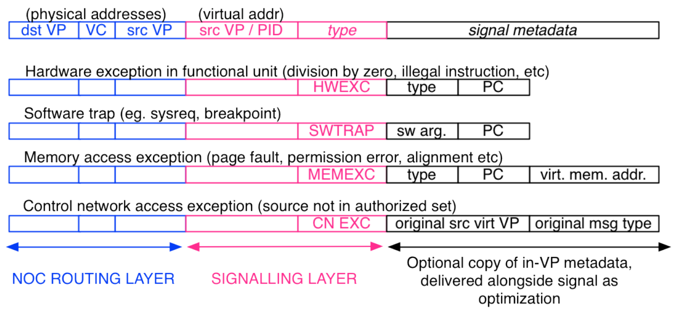
(physical addresses)
(virtual addr)
dst VP
VC
src VP
src VP / PID
type
signal metadata
Hardware exception in functional unit (division by zero, illegal instruction, etc)
HWEXC
type
PC
Software trap (eg. sysreq, breakpoint)
SWTRAP
sw arg.
PC
Memory access exception (page fault, permission error, alignment etc)
MEMEXC
type
PC
virt. mem. addr.
Control network access exception (source not in authorized set)
CN EXC
original src virt VP
original msg type
NOC ROUTING LAYER
SIGNALLING LAYER
Optional copy of in-VP metadata, delivered alongside signal as optimization

Figure 2. Suggested packet format for the SN.

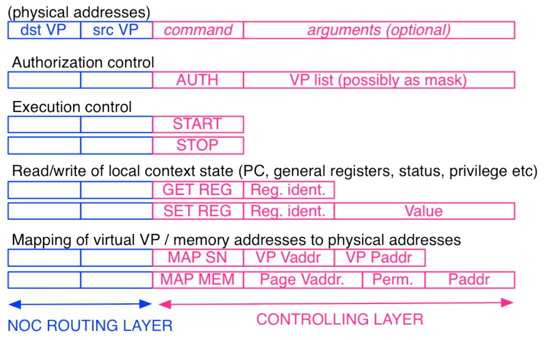

Figure 3. Suggested packet format for the CN.

4.1 Model specification

Taking the physical multiplicity of cores and VPs as a starting point, we *banish privilege levels and automatic context switches entirely*. We do this using two logical networks: a privileged control network (CN) and an unprivileged signalling network (SN), independent from memory, see fig. 1. This capitalizes on the availability of multiple logical NoCs already found in modern multi-cores.

Each VP belongs to a *group*, each group with an address on the CN and SN. Groups capture the reality of "lightweight threads" found in most many-core architectures and which may not be individually addressable. Each group further has a set of authorized VP addresses that the group can receive CN messages from. A VP in a group may not modify its authorized set unless it has its own CN address in the set already. Moreover, the memory address translation table of a VP may only be altered by CN messages. This is the basis for isolation. We then define that the machine fairly schedules all VPs that do not receive CN/SN messages. This abstracts the reality of all many-core architectures already, and guarantees execution independence of system tasks.

The SN enables asynchronous signalling as follows. Each VP may send an arbitrary SN message to a *virtual VP address*. Virtual VP addresses are translated to physical addresses like for memory, using a translation table that can only be configured via CN messages, then routed to the destination VP on the SN. *Some* VPs (at least one) may also *wait* for messages on the SN network, to "listen" for signals from other VPs, and can inspect the source address of all SN messages. Faults and exceptions (including CN authorization rejects) are automatically translated by the machine to an SN message to a pre-defined VP address. Particular implementations of AM^3 may route exception-related messages through the same translation as software-generated messages to make the architecture virtualizable, although this is not strictly required to support a two-level Unix-like OS. Example packet formats are given in fig. 2.

The CN enables system VPs to manage user VPs as follows. The model guarantees that *some* VPs (at least one, a superset of those that can receive SN messages) may send CN messages to a) start and stop the execution b) read and write the internal state c) configure memory/SN address translation and d) configure the CN authorization set, remotely, to an arbitrary VP group identified by a program variable (e.g. in a register). The processing of CN messages is never blocking. Example formats are given in fig. 3.

The queuing properties of the SN and CN networks are left unspecified in this article, although obviously the networks must guarantee delivery. A combination of fixed buffers, per-hop acknowledgements and a proper programming discipline reading SN messages, are sufficient for deadlock freedom and preventing message loss. In this first definition, in-order delivery is required, although we believe the ordering requirement on the SN could be relaxed.

Finally, AM^3 also provides its keystone property: *there are at least 2 VPs in the system*. This is necessary since a single VP running user code cannot be "interrupted".

This keystone, together with the introduction of the CN and SN, are the defining characteristics of AM^3. Although VP configuration messages are rarely found in contemporary software-visible ISAs, they would not require extra hardware since architects already embed this logic for low-level hardware testing and troubleshooting. The SN, and SN address translation are also already implemented in multi-core hardware for virtual interrupt routing. The reception of SN events as explicit waits for new messages instead of VP preemption constitutes a hardware simplification. The CN security filter based on authorized sets is a new feature, however, which does not yet exist in architectures and must be added to support privilege separation *in lieu* of per-core privilege levels.

4.2 Adapting Unix to run on AM^3 platforms

Assuming an architecture presents AM^3 to software instead of preemption and privilege levels, a Unix-like OS kernel must be adapted to the new model as follows.

First a dispatcher must be introduced, running on SN-receiving VPs, which waits for events and invokes the legacy "interrupt handlers" explicitly. The decoupling enabled by an explicit dispatcher also enables dispatching handlers onto different VPs, possibly running in parallel. The memory paging mechanisms are left largely unmodified, and the SN translation tables can be implemented by simply extending the virtual interrupt tables already used in contemporary architectures.

Then the process/thread scheduler must be adapted, so that threads are dispatched to different VPs using CN messages. Since the control flow can only be changed remotely for an entire VP group, the system scheduler must take care that a thread that *may* receive in-thread asynchronous signals ("signal" here refers to the C/POSIX notion) must be the only thread running in its VP group. Or conversely, only use multiple lightweight VPs in a group if all threads allocated to the group are guaranteed to not receive signals.

Then the syscall wrappers in the C library (and equivalent support software for other languages), as well as the stub code to return from a signal handler (including `siglongjmp`), must be adapted to send a SN message and halt the thread; and the syscall entry points in the OS kernel must be adapted to interact with userspace tasks using CN messages, including restarting the remote VP upon completion of a syscall or returning from a signal handler.

The changes are further invisible from application code which may continue to assume the C/POSIX model unchanged. Relinking is necessary however to use the new syscall wrappers, and recompilation may be necessary if the many-core ISA differs from already supported platforms.

4.3 System lifecycle

4.3.1 Initialization

We assume that a platform implementing AM^3 starts in a state where all but one VPs are idle. The VP which is automatically activated to run the boot code must be part of the sender set for the CN. If this VP is also a SN receiver, the boot code can start running the OS kernel directly. Otherwise, the boot code must issue a CN "start" message to a SN-receiving VP to run the kernel there.

The OS kernel then initializes, as usual, I/O devices and management data structures for paging, buffered I/O, networking etc. When the OS kernel is ready to serve applications, another VP is selected to run the "init" process (common parent in Unix) and the CN is then used to set up its initial execution context (registers, memory translation, SN translation) and start process 1. Finally the OS kernel starts its SN dispatcher. To increase throughput during normal system operation, the dispatcher can be replicated on other SN-receiving VPs to share the load of incoming syscalls and other events from the running programs.

4.3.2 Process lifecycle

The lifecycle of all processes (and software threads) starts with an existing process using the `fork` syscall (typically `clone` for threads). When the program issues `fork()`, the syscall wrapper in the C library translates this call to a message on the SN network to the designated OS dispatcher, and then halts the current VP. Upon receiving this message, the OS then uses the CN to retrieve the state of the forking process/thread, prepares a new process/thread context with a copy (page tables, file descriptors, etc.), then uses the CN to initializes another VP with this copy. Then a "start" message is sent to both the original and the new VP to resume execution (fig. 4).

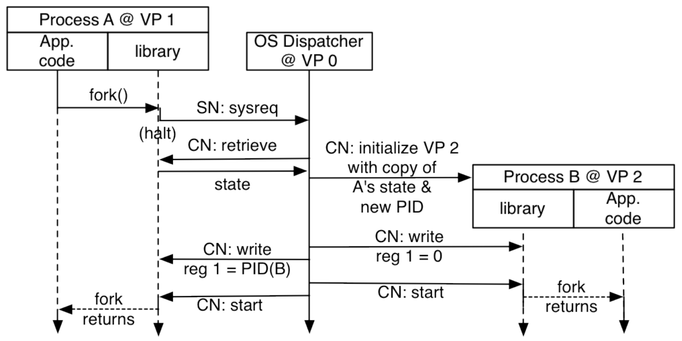

Figure 4. Process creation with AM^3 (and invoking syscalls).

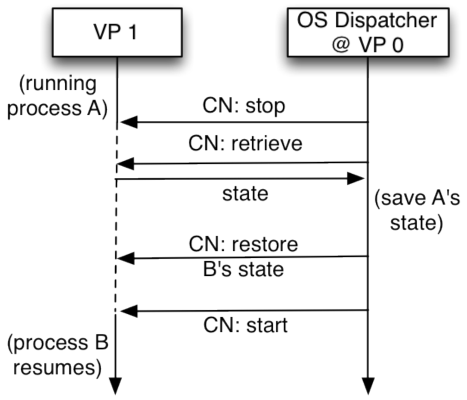

Figure 5. Time sharing in AM^3.

The general case of invoking regular syscalls, already embedded in the use of `fork` above, consists in sending a SN message then halting the current VP (cf. fig. 7 for another example). This naturally extends to sending SN messages without halting the VP, which directly enables additional concurrency between the program and syscall without switching overhead. This can be used advantageously to accelerate the queuing of asynchronous I/O operations.

Time sharing of VPs is required when there are more processes defined than there are VPs in the system. Although we expect this situation to become less common as the number of cores & VPs on chip grows, AM^3 provides a transparent mechanism for this (fig. 5). On each scheduling event, the preemptive scheduler in the OS kernel sends a CN "stop" message to the VP(s) to time share, then uses the CN to retrieve their state, restore the state of another process, and subsequently resume execution. In this case no SN interaction is required.

When a VP triggers a hardware exception or encounters a TLB miss, the AM^3-compatible hardware will halt the VP and send a SN message to the pre-defined handler. The OS dispatcher on that end receives the exception information and determines the appropriate handling. For TLB misses, the paging operation is performed in-kernel and the CN is used to establish a new valid mapping. If a signal is to be delivered, the OS uses the CN to configure the signal handling context. See fig. 9 for a combined example.

When a process terminates as reaction to a VP event (either an explicit signal or `exit` syscall, or unhandled hardware exception), the OS uses the CN to stop other VPs used by the same process. The VPs can then stay idle until they are reused for another process.

A new situation specific to AM^3 occurs when a process halts all its VPs explicitly. In a traditional architecture where the OS scheduler shares VPs with application threads, an explicit "halt" instruction is merely a long-latency no-op, because the VP is woken up at the next preemption clock interrupt. With AM^3, stopped VPs stay stopped until explicitly restarted, so the question arises of what to do. One approach is to consider that halting all VPs is equivalent to a request for termination. In this approach, a monitor thread in the OS must regularly check the state of all VPs and garbage collect terminated processes. Another approach is to consider that halting all VPs is an implicit wait until a wakeup by another process in the system. In this case no particular action is required from the OS.

4.4 Memory models

As many-core processors grow, a new struggle has emerged between software programmers and architects. From the software perspective, a logically shared memory is desirable; for example it has become a standard requirement of the Unix threading model. From the hardware perspective, multi-core cache coherency is expensive in hardware and energy, and savings are attempted by partially dropping support for strong (sequential) global consistency. In practice, various hybrids have been implemented, with strong consistency at the lower cache levels and relaxed consistency (or no consistency) between distant groups of cores.

The exploitation of AM^3 is mostly orthogonal to the memory model(s) supported by a particular multi-core, to the extent that the OS must be aware of the memory topology of the system and map processes to VPs accordingly.

To start, processes that require the appearance of strong consistency between their threads must be mapped to VPs that share a common memory coherency island. Likewise, groups of processes that share memory for inter-process communication must be mapped together. Conversely, if different VPs are connected to distinct memories or non-shared scratchpads, the OS must acknowledge this: read-only shareable memory (eg. code segments) must be duplicated on each memory partition, and migrating a process across memory partitions requires explicit migration of its memory data.

As long as these general guidelines are applied, the memory topology of a particular AM^3 implementation can be made essentially invisible to application software.

5 Proof-of-concept

We have built an example implementation of our model on top of an emerging many-core architecture, the "Microgrid". The Microgrid project, currently led by researchers at the University of Amsterdam, has different goals, namely to demonstrate how a combination of dataflow scheduling and smart thread management in hardware can accelerate data-parallel compute workloads [20], [21]. However, a by-product of this research is an open source, cycle-accurate simulator of Microgrids of configurable sizes called MGSim [22].

MGSim offers us an ideal environment to test our proposal. For one, Microgrid cores do not support thread preemption; moreover, only a limited form of memory sharing is possible between cores: the cache network only supports strong consistency within a L2 cache (between all VPs connected to the same L2 cache) but only a weak form of causal consistency for operations between L2 caches. *As such, Microgrids cannot (yet) run Unix.* However, MGSim and Microgrids support relatively large core counts (hundreds), large-scale hardware multithreading for in-core latency tolerance (tenths of hardware threads per core), local scratchpads, a packet-oriented NoC, backward-compatible userspace ISAs (Alpha/SPARC/MIPS), and virtual address translation using a model already similar to our proposal in the previous section: paging requests (TLB faults/refills) caused by compute cores are signalled to a separate core to be handled asynchronously. The core micro-architecture modeled in MGSim is openly documented [23], [24], [25] and its C compiler is GCC-based and open source, which makes the exploration of architectural changes to many-cores more tractable.

5.1 Existing Microgrid thread management

Microgrids have a relatively fancy threading model with numerous features to optimize raw performance and performance/watt, which are beyond the scope of this article; we focus here on the only two aspects salient to AM^3.

The first is that each physical core supports two kinds of hardware threads. The first is "lightweight" threads (LTs), of which there can be many running at a time (tens), with only few registers each and sharing their identity as a group ("family" in Microgrid parlance, similar to "warps" in CUDA): they must share the same virtual address space, logical process ID and overall execution state (started/stopped). LTs are optimized for fine-grained data parallelism. The second is "general-purpose" threads (GTs), with a full register set and own identity, but there can be only 1-4 GTs running at a time on a core. For the purpose of this article, both LT groups and GTs are candidate substrates to implement VPs.

The second aspect is that VPs can be controlled and configured from other threads or cores. This is done by issuing a `ctl.*` instruction which takes as arguments the target entity (physical core, GT identifier or LT group identifier) and an optional value. Of interest to us, the *run state* of a VP can be queried (`ctl.q`) and changed (`ctl.start`/`ctl.stop`). Changing the state remotely from "running" to "stopped" merely prevents the VP from (re-)entering the hardware schedule queue, so other VPs on the same core can continue to run unaffected. The PC, process ID for address translation and register values of a VP can be queried and updated remotely as well (`ctl.get`/`ctl.set`). At the level of the entire core, the hardware scheduler can be in the state "halted", "active" or "paused", which can be queried or set remotely (`ctl.core.*`): when transitioning from "active" to "paused", started threads remain started but the pipeline is drained. When the state is changed to "halted", running threads are also stopped and need to be explicitly re-started later with `ctl.start`. Control messages already use a dedicated logical NoC.

A security model was already specified for Microgrids [26] however unfortunately it was not yet implemented prior to our work. Instead, we used a simplified model: each physical core is extended with a hidden register containing a "client" identity key and a physical table of 2 "service" keys. When a request (`ctl.*`) is issued from a core, the hardware NoC interface sends that core's client key alongside the request and the remote NoC interface only accepts a control message if the key is present in the remote service table. We choose 2 entries for the service table to support one OS kernel and one debugger. Changing a core's key and service table is also done via control messages. We have extended the MGSim implementation with this security model, as it was sufficient to provide the CN privilege separation described in section 4 at minimal hardware cost, however we consider a discussion about which hardware design provides the best flexibility/cost/performance trade-offs to fall outside of the scope of this article.

5.2 Architectural extensions

Our proof-of-concept is based on two main additions to the Microgrid micro-architecture that extend it with a SN matching the requirements set forth in section 4: signaling logic on every core, and listener logic on fewer cores, where signal-receiving threads are to be run. We have striven to keep the signaling logic lightweight in chip area and energy usage, whereas the listener logic can be more expensive.

The signaling logic is itself split into exception routing circuits for local faults (division by zero, illegal instruction, but also security exceptions when a NoC message could not be delivered), routing logic to translate signals to NoC requests, and the pipeline logic for one new instructions: `sysreq`. This instruction triggers a signal routed using the same rules as local faults as described below, and also stops the VP that executes it without affecting the other VPs on the same core.

Signals have a number, fixed for hardware faults and given in a register for `sysreq`. The signal number is concatenated with the VP's physical thread ID to form an *originator key*. The originator key is then used as index in a *routing table* in memory to obtain a destination address on the NoC, a pair (core ID, channel ID) in our Microgrid implementation. The base pointer for the routing table is stored in a per-core register. A packet containing the originator key, the VP's logical process ID and an optional argument value (e.g. exception details for hardware faults, explicit argument for `sysreq`) is then sent to the destination address via the NoC. This mechanism is not unlike interrupt vectors, except that each signal is mapped to a network address instead of a local PC entry point and no extra circuit is needed to forcefully flush the pipeline and automatically switch the issuing VP's execution context to another task. The logical process ID is packaged with each request as an optimization: although it is possible for the receiving thread to perform a network round-trip to query it remotely, in our envisioned application (Unix system calls) this would be done in nearly all cases so the optimization is warranted to reduce network traffic. To minimize the time overhead of look-ups, we also implemented a dedicated look-aside buffer, although this component could be omitted if the routing table is always local, e.g. in a scratchpad.

On the receiving side, we leverage the Microgrid's existing general-purpose I/O interface [27]. This interface offers memory-mapped access to a configurable (fixed at design-time) number of virtual channels (VCs) on the NoC, with a hardware cost proportional to the number of VCs and the per-VC hardware buffer size. This logic is an optional feature of hardware Microgrid cores: product designers/manufacturers can choose to omit it from some cores, to makes the cores smaller and thus increase core counts or decrease per-core energy usage at a fixed silicon budget. For use as signal delivery mechanism, we propose to implement this circuit in a subset of all cores, for example only one core per memory coherency island on the chip (e.g. one per L2 cache in the Microgrid), with a minimum of 2 VCs per supporting core: one for page faults / TLB refills and one for other signal types. (A separate VC is necessary for translation events to prevent deadlocks when signal handlers use virtual addresses.)

Finally, we also extended the Microgrid MMU to route TLB misses, invalidation and refill events through the same mechanism, but with the originator key and process ID fixed to a value invalid for regular signals (0).

To summarize, our architectural extensions are composed of extra routing logic on each core, one ISA instructions and the memory-mapped I/O logic on "receiving" cores only. Using CACTI simulations [28], we estimate the area increase to not exceed 3% per core on "signaling-only" cores and 11% on "receiving" cores using the same technology parameters as previous Microgrid literature [29], [21].

5.3 Exploitation in software

Since we are defining a Unix hardware accelerator, it should come to no surprise that extremely little work is left to an OS kernel to exploit this hardware and simulate the C/POSIX machine model for application code, as described in section 4.2:

- Unix process threads are mapped to platform VPs as-is: when a thread is known in advance to never receive C signals, it can be ran in a hardware LT (cf. section 5.1), otherwise a hardware GT is used instead. Specific to this architecture, all threads of any given process must be mapped to VPs that share the same L2 cache, as otherwise in-process memory consistency is not ensured;
- syscall wrappers are re-written to use `sysreq`;
- the syscall entry point in the OS kernel is adapted to read from I/O VCs instead;
- the virtual memory interrupt handling routine is also adapted to read paging events from its I/O VC;
- upon start-up, the OS kernel runs its syscall entry point in a dedicated VP on all cores equipped with

the I/O hardware, with routing tables on all other cores configured so as to route signals to the closest management core on the NoC;

- accessing the user context, or communication between userspace and system-space is done via `ctl.get`/`ctl.put`;
- "logical preemption" to support e.g. coarse-grained time sharing and in-process asynchronous signal delivery (C's signals) is also performed over the NoC using `ctl.stop,get,set,start`;
- "return from signal handler" or `siglongjmp` must also use `sysreq` to transfer control, since there is no stack frame to "return" to.

We posit also that the implementation described in this article is fully virtualizable at zero extra hardware cost, since signal routing is memory-based, the VC I/O operations are memory-mapped in their respective's VP's virtual address space, and the privilege domains are virtualizable because security faults are routed as exceptions in a higher-privilege domain. However this was not tested yet and should thus be explored further in future work.

5.4 Evaluation

Our first micro-benchmark is to determine how many cycles it takes for a full round-trip from the point a syscall is issued in a userspace thread to the point the thread resumes normal execution. On conventional hardware (fig. 6) we time batches of 20 invocations to `getpid`. We choose `getpid` as it is one of the syscall with the least amount of work for the operating system (a simple memory lookup). We invoke using `syscall(SYS_getpid)` to avoid the overhead of the standard C library wrapper. N batches are executed, where N is chosen on each platform so that the variability of measurements for the minimum and average time per syscall becomes smaller than 5% (N > 100.000 on all platforms). On the Microgrid (fig. 7) we use a program equivalent to the following:

```
user_main:
   for i = 0 to N:
       ts := TSC() # time stamp counter
       sysreq getpid
       t += TSC() - ts
   print t / N

systemcode:
   repeat:
       vtid := read from VC 0
       pid := pid_table[vtid]
       ctl.put vtid, 0, pid
       ctl.start vtid
```

Our results are reported in table 1. The rightmost columns report times in processor clock cycles. The "Prec." column is the estimated precision of the timestamping facility for the platform. On the x86 ISA, and the Alpha ISA used in MGSim, the timestamping is rather precise: the time stamp counter in hardware can be sampled with a single instruction. On ARM however, the hardware counters cannot be read by user code and we have to rely on a syscall instead. Now, of course, using a syscall would also introduce a sampling error. To account for this, we calibrate our benchmark by first running 100.000 time sampling operations in pairs, and measure the minimum time interval

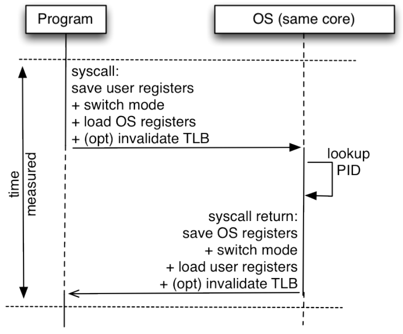

Figure 6. Invoking `getpid` on conventional hardware.

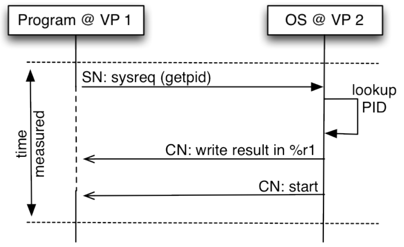

Figure 7. Invoking `getpid` with AM^3.

between all pairs. We name this "precision", the minimum amount of real time that can be reliably detected by the platform's measurement facility. The results in the 3 other columns should thus be interpreted +/- this precision.

As can be seen from the results, with the traditional platforms Unix syscall handling has to traverse software logic before control is handed to the handler and back. Assuming most of that time is spent doing context switches, the cost of these can be estimated from the results (2 switches per syscall). On the simulated Microgrid, the round-trip syscall latency on the same core is 18 cycles, and on adjacent cores 50 cycles due to an extra round-trip network latency of 32 cycles. There is no variability because the simulator is deterministic and no other activity is simulated.

The large values for the maximum column are caused by scheduling artifacts, when the benchmark is interrupted by other tasks and/or the benchmark thread is migrated to another core by the OS. Since the average stays close to the minimum, we can conclude these events are relatively infrequent.

The second micro-benchmark determines how many cycles it takes to deliver a software exception, by means of accessing an invalid pointer. This reflects the cost of context switches in virtualized environments, when privileged operations in a guest OS cause a hardware fault that is redirected back to the guest OS by the hypervisor. In our benchmark we measure the time taken from the point the pointer is accessed to the point control is transferred to the signal handler. On conventional hardware (fig. 8) we use the standard `signal` machinery. On the Microgrid (fig. 9) we use a program equivalent to the following:

```
var end
user_main:
   for i = 0 to N:
       ts := TSC()
       load [0]
```

Table 1
Syscall round-trip time.

Operating system	Processor	Freq (MHz)	Avg (ns)	Min (ns)	Max (ns)	Prec. (ns)	Avg (cc)	Min (cc)	Max (cc)	Prec. (cc)
OS X 10.6 64-bit	Intel Core2 Duo P8600	2400	357.7	343.5	84614.3	0.75	858.4	824.4	203074.2	1.8
Linux 3.4.104 32-bit	Exynos5420 ARMv7l	1900	625.7	541.7	232733.3	68.75	1188.8	1029.1	442193.3	130.6
Linux 3.6.11 32-bit	BCM2708 ARMv6l	700	554.1	299.9	745450.0	49.92	387.8	209.9	521815.0	34.9
Linux 3.2.64 32-bit	Intel P4	2386	188.4	187.2	7489.4	2.01	449.6	446.6	17871.0	4.8F
reeBSD 10.1 64-bit	Intel Atom N2800	1860	347.6	344.7	2417.3	0.75	646.5	641.2	4496.1	1.4
Linux 3.8.12 64bit	AMD Opteron 6172	2100	86.6	86.1	3091.9	1.64	181.8	180.8	6492.9	3.5
Linux 3.13.0 64-bit	Intel Core2 Duo E8335	2667	95.7	92.8	13411.9	0.75	255.3	247.5	35769.5	2.0
Microgrid	OS on different core	1000	50.0	50.0	50.0	1.00	50.0	50.0	50.0	1.0
Microgrid	OS on same core	1000	18.0	18.0	18.0	1.00	18.0	18.0	18.0	1.0

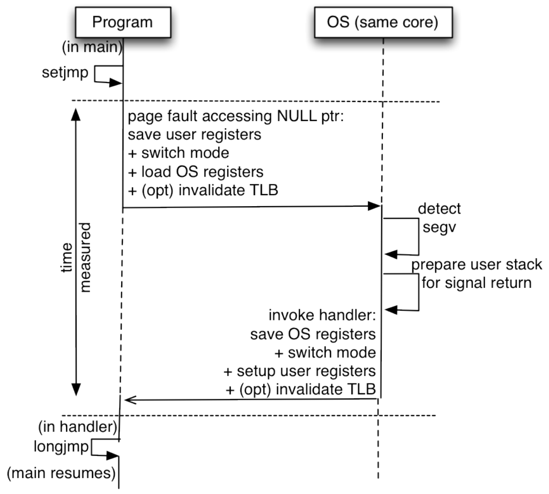

Figure 8. User handling of page faults on conventional hardware.

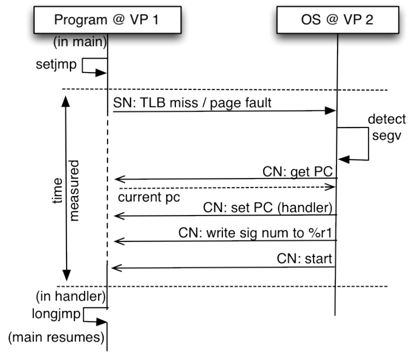

Figure 9. User handling of page faults using AM^3.

```
        t += end - ts
    print t / N

user_handler:
    end := TSC()
    sysreq # (return from empty handler)

systemcode:
    repeat:
        vtid := read from VC 0
        oldpc := ctl.get.pc vtid
        ctl.set.pc vtid, &user_handler
        ctl.put vtid, 0, SIGSEGV
        ctl.start vtid # (resume at handler)
        vtid := read from VC 0
        ctl.set.pc vtid, oldpc
        ctl.start vtid # (resume normal)
```

The results of our measurements are reported in table 2.

Table 3
Signal routing overhead.

Lightweight compute threads	IPC w/ no-op	IPC w/ sysreq	Slowdown
1	0.298	0.291	-1.64%
2	0.371	0.350	-5.59%
5	0.470	0.463	1.14%
10	0.727	0.730	3.84%
20	0.905	0.904	4.32%
50	0.982	0.983	4.46%
100	0.993	0.998	4.46%

The measurements show that page fault handling and signal delivery is significantly more expensive on all platforms than simple syscalls. We attribute this to the additional work required to check the page tables in memory for the missing entries and prepare the signal handler in software. In OS X (Darwin), it is expected that the overheads are relatively higher since Unix signaling is emulated on top of Mach.

To evaluate how much overhead signal routing imparts on normal pipeline processing for compute code, we ran another micro-benchmark that interleaves compute code and signal-generating code on the same core. The compute algorithm used is a Mandelbrot set approximation. This algorithm is computation-heavy (few memory operations), and thus keeps the pipeline busy most of the time. We parallelize it with one lightweight thread per row or block in the complex plane. The micro-benchmark then interleaves the compute threads with a "pertuber" thread on the same core. This thread either runs only no-op instructions, or only `sysreq` instructions, in a loop. When running `sysreq`, a handler on a different core simply resumes execution of the perturber thread. We measure instructions per cycle (IPC), which we take as representative of hardware utilization, and the relative slowdown on total execution time of the computation with a `sysreq` perturber relative to execution time without any perturber. The results are reported in table 3. Since our substrate architecture is single-issue, the maximum possible IPC is 1. This is approximated when more threads interleave in the pipeline while some are waiting on the FPU. Also, since there is no branch predictor, only one instruction from the perturber is present in the pipeline at any time and the maximum IPC imputable to the pertubator thread is 0.125 (for every 8 instructions at IPC = 1, only 1 can come from the pertuber). What the results show is that when there are only 1 or 2 compute threads running, the addition of a perturber thread actually slightly *improves* performance (up to 5.59%). This is because the extra

Table 2
In-process page fault delivery time.

Operating system	Processor	Freq (MHz)	Avg (ns)	Min (ns)	Max (ns)	Prec. (ns)	Avg (cc)	Min (cc)	Max (cc)	Prec. (cc)
OS X 10.6 64-bit	Intel Core2 Duo P8600	2400	12850.0	11801.3	2856101.3	15.00	30840.0	28323.0	6854643.0	36.0
Linux 3.4.104 32-bit	Exynos5420 ARMv7l	1900	7945.5	6625.0	608583.0	1375.00	15096.5	12587.5	1156307.7	2612.5
Linux 3.6.11 32-bit	BCM2708 ARMv6l	700	7450.3	5997.7	3254003.8	998.40	5215.2	4198.4	2277802.7	698.9
Linux 3.2.64 32-bit	Intel P4	2386	4194.7	4140.5	138534.1	40.23	10009.3	9880.0	330564.0	96.0
FreeBSD 10.1 64-bit	Intel Atom N2800	1860	3418.8	3300.5	49861.8	15.05	6359.0	6139.0	92743.0	28.0
Linux 3.8.12 64bit	AMD Opteron 6172	2100	1162.2	1144.8	75511.4	32.38	2440.6	2404.0	158574.0	68.0
Linux 3.13.0 64-bit	Intel Core2 Duo E8335	2667	1086.6	1053.6	23202.1	15.00	2897.9	2810.0	61880.0	40.0
Microgrid	OS on different core	1000	96.0	96.0	96.0	1.00	96.0	96.0	96.0	1.0
Microgrid	OS on same core	1000	65.0	65.0	65.0	1.00	65.0	65.0	65.0	1.0

instructions from the perturber delay compute instructions enough that their operand is ready by the time they are issued, whereas more dataflow misses are incurred without the perturber. When there are more compute threads, the addition of the perturber indeed reduces performance, but only by 4.46% max. Also, the IPC is higher with a `sysreq` perturber than with no-ops because `sysreq` removes the perturber from the schedule queue until it is resumed remotely, so it executes fewer instructions overall.

We also ran an equivalent benchmark using memory-intensive compute threads, running an FFT. The results are equivalent to the previous case and are thus omitted here. However, we also ran a variant that executes the handler thread for `sysreq` *on the same core*. Here, the performance was reduced by approximately 8% for the compute-intensive workload, and 14% for the memory-intensive workload at maximum number of lightweight threads (32). The reason why the memory-intensive workload is more impacted is that the extra memory activity caused by the handler thread in both I-cache and D-cache reduces locality in the computation.

To summarize, the *routing* of software-defined signals has negligible overhead on performance, and *out-of-core signal handling* marries very well with hardware multi-threading to preserve cache locality and reduce performance jitter on unrelated threads.

6 Related and previous work

The idea to leverage an inter-processor network in a Unix-like operating system is not new. Perhaps the closest relative to our proposal, also a direct inspiration for our work, is the CM-5 Connection Machine [30], [31]: this supercomputer consisted of 32 to thousands of SPARC processors, each equipped with some local memory and connected via three separate, high-speed packet-switched networks for data, control and diagnostics [32]. Like our proposed CN, the CM-5 control network could also only be used by the operating system. The control network was organized as a binary fat tree that could be split in sub-trees to partition the system. In each partition one control processor would run the full Unix kernel (CMOST) and serve as partition manager, while other processors in the partition only ran a simple microkernel. The overlap between the CM-5 and our proposal is that the CM-5 can be said to implement AM^3, although it *also* implements traditional interrupt-driven syscalls and time sharing on each processor (since each node was a fully fledged SPARC). The insight here is that the interrupt handling circuits on the CM-5 processors could be dropped and the whole supercomputer could still run Unix as described in the present article. Also, the CM-5 could only provide isolation using a binary partition of the tree, whereas our model supports arbitrary partitions.

In more recent work, two OS research projects have embraced the many-core revolution from a different angle: MIT's fos [33] and ETH/Microsoft's Barrelfish [34]. Both fos and Barrelfish exist in the same research domain: exploring new OS design directions to leverage many-core chip resources more safely, robustly and efficiently. The focus of Barrelfish lies on managing on-chip resource heterogeneity, whereas fos focuses on scalability (with Clouds of many-cores as envisioned target platform). Like in our model, both embraces platform parallelism and are designed from the ground up to try and run application and OS code on different cores. Interactions between application components and with OS code is also done via message passing. Shared memory can be exploited if available in the architecture but is not required by the OS to function. However, the known implementations of fos and Barrelfish still target current commodity processors and thus emulate message passing using shared memory, instead of exploiting on-chip NoCs directly. Moreover, both projects are the output of research in OS design, and C/POSIX compatibility does not appear to be a strong requirement—it is envisioned as a userspace compatibility layer in Barrelfish, and not discussed at all in fos literature.

A more direct competitor to our proposal has emerged in recent GPGPU accelerator offerings from NVidia. From what is visible in the CUDA specifications [35], NVidia's devices do not yet support sufficient functionality regarding privilege separation and remote scheduling control to enable running arbitrary Unix threads on the GPGPU device. However, NVidia's recent products and latest CUDA releases allow "kernel" code running on the many-core accelerator to perform calls to C library functions like `printf`, and the effects thereof are written to a local memory which can be retrieved asynchronously by the host processor to present the kernel's output. The CUDA documentation still explicits that only a limited subset of the C library is available to kernel code in this way, and to our knowledge CUDA platforms do not yet offer a generalized mechanism to invoke system services from the GPU cores.

7 Discussion and future work

Two questions were raised by our community while working on this article. The first is is whether our focus on C/POSIX is perhaps too dismissive of recent results in OS research. The second is how we intend to sell our work to the main industry players like Intel, ARM or NVidia. We answer these in sections 7.1 and 7.2. We then outline in section 7.3 a few additional findings that resulted from our proof-of-concept, providing directions for future work.

7.1 Is AM^3 specific to Unix?

The original motivation for this work was to evaluate empirically the claims to generality that had been made in previous Microgrid-related literature. With our background in architecture and systems programming from the POSIX perspective, "generality" could only mean "runs Unix", but the Microgrid was disappointingly lacking in this regard. The work described in this article initially resulted from "scratching the hacker's itch", i.e. seeking the highly-regarded, often sought-after thrill of first booting a Unix shell on a new platform. Our work is, in a way, a reminder of the symbiotic relationship between Unix OS developers and processor architects, with the regularly renewed interest of one side to support the work of the other side acting as long-term stabilizing factor for research in both architecture and OS design. However, many processor architects' heads are now being scratched to reconcile the need for more on-chip parallelism with legacy software compatibility. This is a new challenge, and AM^3 provides a way to smoothly drop some requirements from architectures without losing Unix, and thus preserve the beneficial symbiosis, while gaining NoC awareness as a new feature.

Yet we could consider AM^3 without considering Unix compatibility at all. After all, AM^3 is merely a hardware/software interface that optimizes the practical implementation of any software stack built around the Actor [36], [37], CSP [38] or π-calculus [39], [40] abstractions. An AM^3-compatible platform could, for example, be used as a direct hardware implementation of the Actor-based Erlang [41] virtual machine. This would not need Unix abstractions since an Erlang applications usually is, in practice, the sole occupant of its underlying hardware platform. Similarly, AM^3 could be directly used as a lightweight back-end for SAC [42], Chapel [43] or X10 [44] without changes to the existing language semantics, providing access to many-core accelerators to existing HPC code. Although our own work was limited so far to porting FreeBSD [45] to the Microgrid, we are in contact with multiple researchers in parallel programming language design who have expressed interest in our approach and helped generalize the AM^3 abstractions.

7.2 Incentives in the current architectural landscape

AM^3 could mean different things to, say, Intel and NVidia. We summarize this in fig. 10. For people with a background where many-cores result from putting multiple single cores together, with compatibility with legacy code as a chief concern, AM^3 provides a route to reduce the complexity per core, and thus increase core counts on chip and energy efficiency, without losing the software compatibility, and adding better support for on-chip networks. For those hoping many-cores will be the key to continued performance increases, AM^3 greatly extends the programming model and provides immediate compatibility with most legacy software, at limited costs in per-core chip area and overall performance.

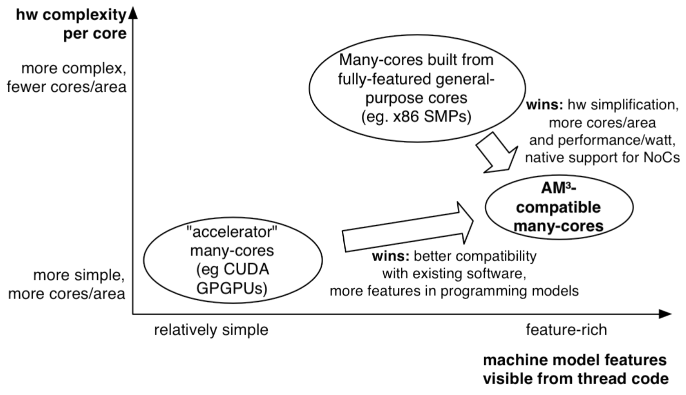
hw complexity per core
more complex, fewer cores/area
more simple, more cores/area
Many-cores built from fully-featured general-purpose cores (eg. x86 SMPs)
wins: hw simplification, more cores/area and performance/watt, native support for NoCs
AM³-compatible many-cores
"accelerator" many-cores (eg CUDA GPGPUs)
wins: better compatibility with existing software, more features in programming models
relatively simple
feature-rich
machine model features visible from thread code

Figure 10. Incentives for current many-core providers.

7.3 Ancillary discoveries and future work

While contemplating the conclusions of section 5.4, namely that running system functions on the same core as compute code can reduce cache locality, we started searching for other applications of this result. Our own background in programming language design and implementation led us to three direct beneficiaries: asynchronous I/O, memory managers (MMs: allocators and garbage collectors), and functional reactive programming (FRP).

Asynchronous I/O is perhaps the most trivial application, since a VP can emit a request to another VP via the SN and continue to run asynchronously. Although not yet available in our proof-of-concept implementation, a "`sysreq.a`" instruction could send a syscall request but leave the VP running. This could then be used in C compilers to optimize multiple successive calls to `memcpy` to run them in parallel.

Userspace MMs are interesting because they have historically been already extremely well encapsulated as a request-response API: application code places a request for memory, and the manager responds with a pointer or a failure signal; there is no state shared between application code and MMs other than the managed memory areas. Thanks to this, we were able to segregate MM code in our ported C library to run on a dedicated core, receiving requests via `sysreq` from all VPs in the local L2 cache cluster. This enabled measurable reductions of cache misses on the MM data structures in our benchmarks. As an unexpected benefit, running the MM on a separate core enabled us to trace the MM code without any overhead to the client compute code running on other cores, because the MM can deliver its results asynchronously to a client MM before emitting its tracing event. Although we could not yet demonstrate this, our preliminary work on GC code strongly suggests that the overhead of stopping threads during the mark phase becomes orders of magnitude lower when using CM "stop" and "start " events, leaving the state of VPs in their respective hardware cores untouched during the GC run.

"Functional reactive programming" [46], [47] is a software pattern to structure applications as functional equations of behavior over time and event variables. The attractiveness of this pattern comes from the rich compositionality and brevity of FRP programs. A salient feature of FRP is that the programs are expressed as a finitely sized dataflow graphs. By mapping each FRP behavioral equation to an

AM^3 VP that can listen on the SN, and assigning each FRP signal to an SN channel, we could run an entire FRP task network in parallel, using the SN as synchronization and scheduling substrate, with no model simulation overhead. Future work could thus consider extending functional languages to use AM^3 as FRP accelerators with limited effort. This opportunity extends, in principle, to any dataflow-like programming model, although the stateless nature of FRP equations and relative rigidity of the typical FRP task graph makes the mapping particularly seamless.

8 Conclusions

Providers of new computer architectures regularly try to introduce new programming models, e.g. recently CUDA/OpenCL for GPGPUs, and displace C/POSIX as the control interface to platform parallelism. However this is unlikely to succeed. For one, the I/O hardware market has created a reinforcement loop: OS and language support are written around C/POSIX, so new drivers needs to be compatible with C/POSIX, but drivers are expensive to change so they are long-lived, so new OS and language versions remain compatible with C/POSIX, etc. Second, because the stacking of multiple programming models to manage parallelism (C/POSIX for "host" code, custom for "accelerator" code) creates extra system complexity, e.g. to control sharing of an accelerator between multiple processes, which in turn drives down system performance and/or energy efficiency. If a single interface is desirable, and if we are stuck with C/POSIX by external factors, we may as well try and adapt C/POSIX instead to effectively embrace many-cores.

This article presents the results of a step in this direction, performed while exploring how to leverage architectural features found in contemporary many-core architectures to accelerate the process-system interface in the C/POSIX machine and programming models.

Our first contribution is a general analysis of how the machine model underlying C/POSIX operating systems can be revisited to take advantage of networks-on-chip and platform parallelism. The result is a new abstract many-core machine model, AM^3, which features two inter-core networks but no interrupts, and which is rich enough to support C/POSIX.

Our second contribution is a a proof-of-concept implementation of our proposal. Our proof-of-concept is based on an existing research-grade many-core architecture originally designed to maximize performance and energy efficiency but without direct support for C/POSIX; we were able to add support for C/POSIX in that platform using our model with only minimal architectural changes; the overhead of calls to OS functions from application code then becomes multiple orders of magnitude smaller than in contemporary architectures.

The third contribution of this article is a discussion of the indirect benefits of this approach. Beyond the reduction in hardware complexity made possibly by dropping the traditional circuits in charge of privilege separation and context switching, which would in turn drive down per-chip manufacturing costs and energy efficiency, the execution of system tasks in different hardware threads or even cores enables higher utilization of the core pipeline and thus higher overall performance. When system tasks are split in different cores, memory locality is improved and cache hit rates increase, also contributing to performance and efficiency improvements. As an indirect benefit, using native hardware support for system messaging enables simpler run-time systems for parallel programming languages, which may in turn enable new programming languages or programming models for emerging many-cores.

References

[1] S. L. Peyton-Jones, "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine," *Journal of Functional Programming*, vol. 2, pp. 127–202, 3 1992. [Online]. Available: http://journals.cambridge.org/article_S0956796800000319

[2] A. Voellmy, J. Wang, P. Hudak, and K. Yamamoto, "Mio: A high-performance multicore IO manager for GHC," in *Proc. 2013 ACM SIGPLAN Haskell Symposium*, ser. Haskell '13, Boston, MA, USA, September 2013, (to appear).

[3] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, March 1995.

[4] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus IPC: the end of the road for conventional microarchitectures," *SIGARCH Comput. Archit. News*, vol. 28, pp. 248–259, May 2000.

[5] R. Ronen, A. Mendelson, K. Lai, S.-L. Lu, F. Pollack, and J. Shen, "Coming challenges in microarchitecture and architecture," *Proceedings of the IEEE*, vol. 89, no. 3, pp. 325–340, mar 2001.

[6] F. J. Pollack, "New microarchitecture challenges in the coming generations of CMOS process technologies (keynote address)," in *Proc. 32nd annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 32. Washington, DC, USA: IEEE Computer Society, 1999.

[7] J. E. Thornton, "Parallel operation in the Control Data 6600," in *Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*, ser. AFIPS '64 (Fall, part II). New York, NY, USA: ACM, 1965, pp. 33–40.

[8] B. Smith, "Architecture and applications of the HEP multiprocessor computer system," *Proc. SPIE Int. Soc. Opt. Eng.; (United States)*, vol. 298, pp. 241–248, 1981.

[9] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," *SIGARCH Comput. Archit. News*, vol. 23, pp. 392–403, May 1995.

[10] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-Threading technology architecture and microarchitecture," *Intel Technology Journal*, vol. 6, no. 1, pp. 1–12, 2002. [Online]. Available: http://www.mendeley.com/research/hyperthreading-technology-architecture-and-microarchitecture/

[11] International Standards Organization and International Electrotechnical Commission, *ISO/IEC 9899:2011, Programming Languages – C*, 1st ed. 11 West 42nd Street, New York, New York 1O036: American National Standards Institute (ANSI), December 2011. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/

[12] IEEE Standards Association, *IEEE Std. 1003.1-2008, Information Technology – Portable Operating System Interface (POSIX®)*. IEEE, 2008.

[13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, OCtober 2003.

[14] P. Vaibhav. (2009, June) Reworking the call out API: towards a tickless kernel. [Online]. Available: https://wiki.freebsd.org/SOC2009PrashantVaibhav

[15] H. Akkan, M. Lang, and L. M. Liebrock, "Stepping towards noiseless linux environment," in *Proceedings of the 2Nd International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '12. New York, NY, USA: ACM, 2012, pp. 7:1–7:7.

[16] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *Proceedings of the tenth international symposium on Hardware/software codesign*, ser. CODES '02. New York, NY, USA: ACM, 2002, pp. 73–78.

[17] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, 2002, pp. 409–415.

[18] W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proc. Design Automation Conference, 2001.*, 2001, pp. 684–689.

[19] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core SCC processor: the programmer's view," in *Proc. 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[20] K. Bousias, L. Guang, C. Jesshope, and M. Lankamp, "Implementation and evaluation of a microthread architecture," *Journal of Systems Architecture*, vol. 55, no. 3, pp. 149–161, 2008.

[21] R. Poss, M. Lankamp, Q. Yang, J. Fu, M. W. van Tol, I. Uddin, and C. Jesshope, "Apple-CORE: harnessing general-purpose many-cores with hardware concurrency management," *Microprocessors and Microsystems*, vol. 37, no. 8, pp. 1090–1101, November 2013.

[22] R. Poss, M. Lankamp, Q. Yang, J. Fu, I. Uddin, and C. Jesshope, "MGSim—a simulation environment for multi-core research and education," in *Proc. Intl. Conf. on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS XIII)*. IEEE, July 2013, pp. 80–87.

[23] M. A. Hicks, M. W. van Tol, and C. R. Jesshope, "Towards Scalable I/O on a Many-core Architecture," in *International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS)*. IEEE, July 2010, pp. 341–348.

[24] R. Poss, "On the realizability of hardware microthreading—revisting the general-purpose processor interface: consequences and challenges," Ph.D. dissertation, University of Amsterdam, September 2012. [Online]. Available: http://www.raphael.poss.name/on-the-realizability-of-hardware-microthreading/

[25] M. Lankamp, R. Poss, Q. Yang, J. Fu, I. Uddin, and C. R. Jesshope, "MGSim—simulation tools for multi-core processor architectures," University of Amsterdam, Tech. Rep. arXiv:1302.1390v1 [cs.AR], February 2013. [Online]. Available: http://arxiv.org/abs/1302.1390

[26] J. Masters, M. Lankamp, C. Jesshope, R. Poss, and E. Hielscher, "Report on memory protection in microthreaded processors, Apple-CORE deliverable D5.2," December 2008. [Online]. Available: http://apple-core.info/research.html

[27] M. Lankamp, M. W. van Tol, C. Jesshope, and R. Poss, "Hardware I/O interface on the Microgrid," University of Amsterdam, Tech. Rep. [mgsim14], May 2011. [Online]. Available: https://notes.svp-home.org/mgsim14.html

[28] S. Wilton and N. Jouppi, "Cacti: an enhanced cache access and cycle time model," *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 5, pp. 677–688, may 1996.

[29] R. Poss, M. Lankamp, Q. Yang, J. Fu, M. W. van Tol, and C. Jesshope, "Apple-CORE: Microgrids of SVP cores (invited paper)," in *Proc. 15th Euromicro Conference on Digital System Design (DSD 2012)*, S. Niar, Ed. IEEE Computer Society, September 2012.

[30] J. Palmer and J. Steele, G.L., "Connection machine model cm-5 system overview," in *Proc 4th Sumposium on the Frontiers of Massively Parallel Computation*. IEEE, Oct 1992, pp. 474–483.

[31] W. D. Hillis and L. W. Tucker, "The CM-5 Connection Machine: A scalable supercomputer," *Commun. ACM*, vol. 36, no. 11, pp. 31–40, November 1993.

[32] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak, "The network architecture of the Connection Machine CM-5 (extended abstract)," in *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '92. New York, NY, USA: ACM, 1992, pp. 272–285.

[33] D. Wentzlaff, C. Gruenwald, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal, "A unified operating system for clouds and manycore: fos," Computer Science and Artificial Intelligence Lab, MIT, Tech. Rep. MIT-CSAIL-TR-2009-059, November 2009.

[34] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs, "Embracing diversity in the Barrelfish manycore operating system," in *Proceedings of the Workshop on Managed Many-Core Systems*, June 2008.

[35] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture," in *Proc. 6th international symposium on Memory management (ISMM '07)*. New York, NY, USA: ACM, 2007, pp. 103–104.

[36] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *Proc. 3rd International Joint Conference on Artificial Intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[37] G. A. Agha, "ACTORS: A model of concurrent computation in distributed systems," Massachusetts Institute of Technology, AITR 844, 1985. [Online]. Available: http://dspace.mit.edu/handle/1721.1/6952

[38] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666–677, August 1978.

[39] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, I," *Information and Computation*, vol. 100, no. 1, pp. 1–40, 1992.

[40] ——, "A calculus of mobile processes, II," *Information and Computation*, vol. 100, no. 1, pp. 41–77, 1992.

[41] J. Armstrong, R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in ERLANG*, 2nd ed. Prentice Hall, 1996.

[42] C. Grelck and S.-B. Scholz, "SAC: a functional array language for efficient multi-threaded execution," *International Journal of Parallel Programming*, vol. 34, no. 4, pp. 383–427, August 2006.

[43] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, August 2007.

[44] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005, pp. 519–538.

[45] M. K. McKusick and G. V. Neville-Neil, *Design And Implementation Of The FreeBSD Operating System*. Addison Wesley, 2004.

[46] C. Elliott and P. Hudak, "Functional reactive animation," in *Proc. 2nd ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '97. New York, NY, USA: ACM, 1997, pp. 263–273.

[47] H. Nilsson, A. Courtney, and J. Peterson, "Functional reactive programming, continued," in *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, ser. Haskell '02. New York, NY, USA: ACM, 2002, pp. 51–64.

PLACE PHOTO HERE

Raphael Poss is an assistant professor at the joint Informatics Department of the Vrije Universiteit Amsterdam and the University of Amsterdam, in the Netherlands. He obtained his PhD in computer architecture in 2011 at the University of Amsterdam, under supervision of prof Chris Jesshope. He currently conducts research in hardware/software co-design, new semantic models for future programming languages for large parallel systems and new operating systems for heterogeneous architectures.

PLACE PHOTO HERE

Koen Koning is a PhD candidate at the Vrije Universiteit Amsterdam, the Netherlands. He obtained his MSc in computer science at the Vrije Universiteit, and his BSc at the University of Amsterdam. His research interest lie in the design and implementation of computer systems.