Cutting Out the Middleman: OS-Level Support for X10 Activities

Manuel Mohr Sebastian Buchwald
Andreas Zwinkau
Karlsruhe Institute of Technology, Germany
{manuel.mohr,sebastian.buchwald,zwinkau}@kit.edu

Christoph Erhardt Benjamin Oechslein
Jens Schedel Daniel Lohmann
FAU Erlangen-Nuremberg, Germany
{erhardt,oechslein,schedel,lohmann}@cs.fau.de

Abstract

In the X10 language, computations are modeled as lightweight threads called *activities*. Since most operating systems only offer relatively heavyweight kernel-level threads, the X10 runtime system implements a user-space scheduler to map activities to operating-system threads in a many-to-one fashion. This approach can lead to suboptimal scheduling decisions or synchronization overhead. In this paper, we present an alternative X10 runtime system that targets OctoPOS, an operating system designed from the ground up for highly parallel workloads on PGAS architectures. OctoPOS offers an unconventional execution model based on *i*-lets, lightweight self-contained units of computation with (mostly) run-to-completion semantics that can be dispatched very efficiently. We are able to do a 1-to-1 mapping of X10 activities to *i*-lets, which results in a slim runtime system, avoiding the need for user-level scheduling and its costs. We perform microbenchmarks on a prototype many-core hardware architecture and show that our system needs fewer than 2000 clock cycles to spawn local and remote activities.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming; D.3.3 [*Software*]: Programming Languages—Language Constructs and Features; D.4.1 [*Software*]: Operating Systems—Process Management; D.4.4 [*Software*]: Operating Systems—Communications Management

Keywords X10, Runtime System, Operating System, Scheduling, Many-Core Architecture, Invasive Computing

1. Introduction

The X10 philosophy wants the programmer to express the program with a large amount of parallelism. It provides the concept of an *activity*, which is colloquially described as a lightweight thread. Lightweight means that the programmer does not need to be wary of creating too many activities. This implies that the runtime is required to schedule activities to the corresponding kernel-level threads in a many-to-one fashion. Creating lots of activities gives a lot of freedom to the scheduler. Intuitively, the number of threads should be the same as the parallelism the hardware provides. To minimize synchronization between threads, each thread manages its own pool of activities. Work stealing is employed when load balancing becomes necessary.

This approach has the well-known downside that a blocking call into the operating system (e.g., for I/O) blocks the thread with all the activities it manages. This means that a core might idle although there are activities that could be executed there. The X10 runtime works around this problem by starting additional threads before blocking operations. This implicates some overhead for creating and terminating threads to keep the number approximate to the hardware parallelism. Another problem is that starting an additional thread is easily forgotten when bindings for blocking operations are implemented. If it happens outside the runtime library, an application programmer might not even know that this would be necessary.

To avoid overhead and reduce susceptibility to bugs, we explore the alternative to integrate activity management directly into the OctoPOS [16] operating system (OS), which has been designed for highly parallel workloads on PGAS architectures. Since activities are more lightweight than kernel-level threads, the system becomes simpler and more efficient. In short, this paper presents

- how we implemented activity management by mapping it directly to OS mechanisms in OctoPOS,
- how this simplifies runtime system and OS,
- and an evaluation of the efficiency by measuring running times of various operations on a prototype many-core hardware architecture.

In Section 2, we describe the hardware architecture that OctoPOS is designed for. Then, Section 3 shows the provided OS primitives. In Section 4, we detail our runtime-system implementation. Afterwards, Section 5 evaluates the efficiency and Section 6 gives an overview of related work.

2. Tiled Many-Core Architectures

Future many-core architectures are expected to integrate hundreds or even thousands of cores on a single chip. With such high core counts, maintaining cache coherence across all cores of the system while keeping performance up becomes a major obstacle, as existing cache-coherence protocols do not scale well or considerably increase latencies. One possible solution are *tiled* many-core architectures [10]. They provide cache coherence only for groups of cores and offer a simple scalable interconnection for message-based communication between core groups.

Figure 1 shows an example of a tiled many-core architecture. The building block of such an architecture is a *tile*. In the example,

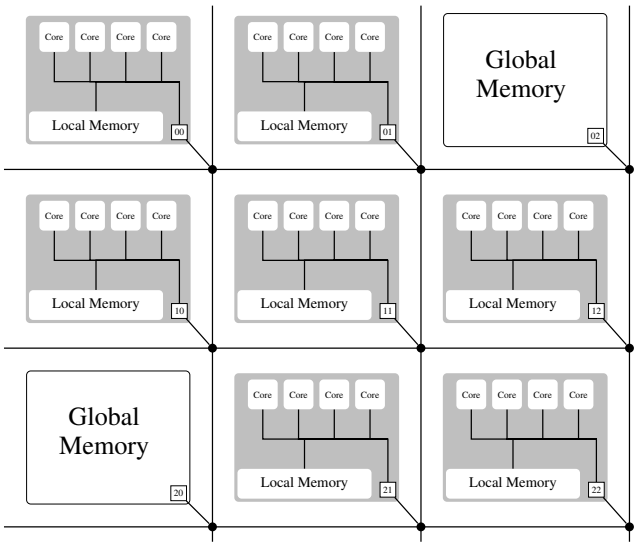

Figure 1. An example of a tiled many-core architecture. The example shown has nine tiles, arranged in a 3×3 mesh and connected by a network-on-chip. Seven of the tiles contain processor cores and two tiles contain external memory.

a tile is a group of four processor cores that share some resources, such as an L2 cache or a small tile-local memory. Most importantly, cache coherence is guaranteed between the cores of a tile. Hence, from a programmer's point of view, a single tile behaves exactly like a common shared-memory multiprocessor system. In general, the number of cores in a tile must be low enough that traditional coherence protocols, such as bus snooping, are still applicable.

Multiple such tiles can be combined to create a tiled many-core architecture, where the tiles are connected via a network-on-chip. Tiles can be heterogeneous in nature and, for example, contain memory instead of processor cores. In such an architecture, *no* cache coherence is provided between the cores of different tiles. Hence, applications spanning multiple tiles cannot be programmed like on a shared-memory system anymore. Partitioning the global memory and thus applying a PGAS programming model becomes the most viable method to program the system. Therefore, X10 is a perfect fit and maps naturally to these architectures: Each tile is presented as a separate place to the programmer. As X10 semantics demand that data must be copied when switching places via `at`, the missing cache coherence across tile boundaries does not cause problems.

3. The Execution Model of OctoPOS

OctoPOS is an operating system designed specifically for PGAS architectures as described in the previous section. Its primary design goal is to exploit fine-grained parallelism in applications directly on the operating-system level. Therefore, OctoPOS offers an execution model that is more lightweight than the "traditional" UNIX process/thread model. The principal idea is that parallelizable control flows are represented in the operating system not as coarse-grained, long-running threads with preemption, but as short snippets of code we call *i*-lets. An *i*-let consists of two components:

- a pointer to a function to be executed, and
- a piece of data passed as argument.

A typical parallel application running on top of OctoPOS splits its work into many packages, creates an *i*-let for each work package, and hands these *i*-lets to the operating system for execution. The OS scheduler distributes the *i*-lets to the CPU cores available, where they will be processed sequentially. Like user-level fibers, *i*-lets use cooperative scheduling. For *i*-lets that run to completion, creation and dispatching are very efficient because the respective execution contexts (i.e., stacks) can simply be reused. Only if an *i*-let performs a blocking operation does the OS have to perform a costlier context switch.

OctoPOS enforces spatial separation of concurrently running applications – that is, each application has its own, dedicated set of CPUs. This is based on the assumption that the many-core system is sufficiently large to accommodate the needs of all applications involved. An OS-level resource manager [12] governs the distribution of execution resources and decides on the concrete CPU mapping at runtime. As a consequence, no preemption is necessary. Applications are granted full control over their core set anyway.

On a system with a tiled architecture, each tile runs a separate instance of the operating system. For distributed applications, code must be shared globally or loaded at the same base address on all tiles. Such applications can communicate using two OS-level primitives.

Remote* i-*let spawning. Code execution on a remote tile is triggered by sending a fixed-size packet over the network-on-chip. The packet consists of a pointer to an *i*-let function plus up to two data words for passing arguments, which can hold by-value arguments or pointers in case of larger input data (transmitted separately). On the receiving side, the *i*-let is inserted into the regular scheduling queue and executed asynchronously to the sender's control flow.

Push-DMA transfer. To allow transferring larger chunks of data between tiles, OctoPOS offers a *push* mechanism that allows copying an arbitrarily large contiguous memory region to a buffer in another tile's local memory. The receiving tile is guaranteed to have a cache-coherent view of the destination buffer after the transfer has completed. Using appropriate hardware support by the on-chip network, the operation is performed asynchronously as a DMA transfer, allowing the sending process to continue work without blocking. The caller of a push-DMA operation can optionally pass a pair of *i*-lets along with the data:

- The first *i*-let will be executed on the sending tile once the operation has completed, which can be used for releasing the source buffer or for implementing custom blocking if desired.
- The second *i*-let will be spawned on the receiving tile, where it can begin processing the transferred data.

For the synchronization of *i*-lets, OctoPOS offers a lightweight barrier-like concept called *signal* which is optimized for a fork-join scenario. The standard pattern in this scenario is one *i*-let that spawns multiple other *i*-lets for parallel work, and then waits for their termination. An OctoPOS signal is initialized with a counter value equal to the number of jobs. After creating the jobs, the spawning *i*-let invokes the `wait()` primitive, which blocks until the counter reaches zero. Each job does its work and afterwards calls `signal()`, which decrements the counter by one. If the number of jobs is not known in advance, `add_signalers()` can be called for new *i*-lets created dynamically to increment the counter.

OctoPOS signals are similar to blocking semaphores, but more lightweight: Only a single *i*-let per signal is allowed to wait, so there is no need for a waiting queue. Activities that were spawned on another tile can signal back to their original tile by sending an *i*-let that performs the signaling.

In summary, OctoPOS is an operating system for PGAS architectures that implements a lightweight *i*-let-based execution model and offers asynchronous operations for cross-tile data transfers and activity spawning.

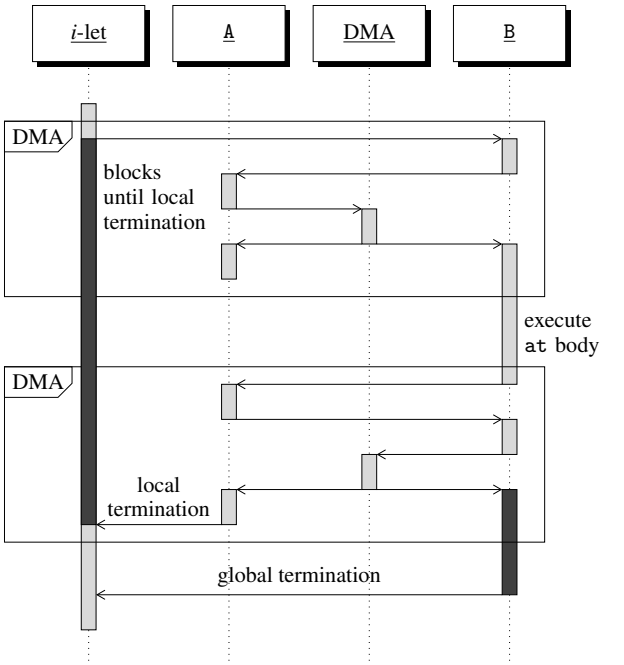

Figure 2. Sequence diagram for an `at` expression. It shows one DMA transfer from `A` to `B` for the values of referenced outer variables and another one from `B` to `A` for the returned value. DMA transfers require a forth-and-back communication to set up a buffer. Dark blocks indicate blocked state.

4. X10 Runtime-System Adaptation

In this section, we describe the mapping of the X10 runtime system to the primitives provided by OctoPOS. First of all, we map X10 activities directly to *i*-lets. Hence, each `async` block in an X10 program corresponds to the creation of an *i*-let, which is then handed to OctoPOS for execution. For the `finish` statement, we use the signal primitives provided by OctoPOS.

The mapping for an `at` expression is more complex, since it requires inter-place communication. Let us look at the expression `at (B) E` which is executed on place `A`. X10 semantics dictate that the values of all variables referred to by `E` must be copied from `A` to `B` before `E` is executed on `B` [18, §13.3.1ff]. The X10 compiler conservatively approximates at compile-time the set of variables used. At run-time, the values of this set are serialized into a buffer. Hence, essentially, a sequential block of memory must be transferred from `A` to `B` before `E` can be executed.

This means that in general, the two data words of an *i*-let are not enough to hold this data and DMA transfers have to be used instead. In the following, we will therefore discuss how an `at` expression can be implemented using the OS interfaces presented. Figure 2 shows the full control flow for an `at` expression.

Before we can set up a DMA transfer, we must allocate the destination buffer on place `B`. The buffer size is only known at run-time as it depends on which values are reachable through the variables that `E` refers to. Hence, we spawn a remote *i*-let on `B`, allocate the buffer and pass the address back to `A` via another remote *i*-let spawning.[1]

Now, on `A`, we can initiate the push-DMA transfer from the source buffer on `A` to the destination buffer on `B`. We specify two *i*-lets to execute once the data transfer has finished: one on `A` to free the source buffer, and one on `B` to perform setup and execution of `E`. After `E` has terminated locally [18, §14], the result of the `at` expression must be transferred back to `A`. This is done using a DMA transfer analogously to the initial transfer.

Hence, we first switch to `A` to allocate a buffer and then pass the address back to `B` to initiate the DMA transfer on `B`. Again, we specify two *i*-lets: One on `B` (the sending side in this case) to wait for global termination [18, §14] of `E` and then signal it to `A`. We also specify an *i*-let to execute on `A` (the receiving side) to notify the original *i*-let, which executed the `at` expression, of the local termination of `E` and pass it the value of `E`, which is the result of the `at` expression.

Note that only the *i*-let that initially executes the `at` expression and the *i*-let that waits for global termination block. All other *i*-lets run to completion. This is achieved by storing the necessary information for a specific `at` invocation in context structures on `A` and `B`, and passing pointers to these contexts back and forth between places via the available data words of an *i*-let.

Special cases. For `at` statements, the second DMA transfer for the result can be omitted. Local and global termination are handled in the same way as for the presented expression case.

For `at (B) async S` statements, the runtime system can assume that the activity on `B` immediately terminates locally. Hence, in this case, once the DMA transfer is triggered, but has not necessarily finished yet, we can already notify the original *i*-let of local termination. Once `S` has terminated globally, global termination is signaled back to `A`.

Moreover, if `S` does not refer to any outer variables and thus no additional data must be transferred to `B`, the runtime implementation is even more efficient. In this case, no DMA transfer is needed and `at (B) async S` is mapped directly to the spawning of a single remote *i*-let. Again, as we know that the remote activity immediately terminates locally and the spawning of a remote *i*-let is a non-blocking operation, the runtime procedure can directly return, and execution of the X10 program can continue.

5. Evaluation

In this section, we analyze both the raw performance of our X10 runtime system as well as its overhead compared to the direct usage of the operating-system C interface. To this end, we measure the running times of multiple X10 programs and compare these times to the running times of semantically equivalent C programs. We measure the running time of executing a local activity (`async` statement), a remote asynchronous activity (`at async` statement), a remote synchronous activity[2] (`at` statement) and a remote synchronous activity returning a value (`at` expression).

Limitations. Before we describe our evaluation setup and discuss the measurements, we explain the limitations of our evaluation. It would be desirable to compare our system to the existing unmodified X10 implementation. However, we changed three parameters in our system: the runtime implementation, the operating system, and the underlying hardware. Hence, only having a common hardware platform would enable a fair comparison, as it does not make sense to compare programs using the original X10 runtime implementation running on Linux on standard hardware with programs that use our modified runtime implementation and run on OctoPOS on our custom many-core architecture.

Unfortunately, this comparison is not possible yet. As Linux has not been ported to our custom many-core platform, running OctoPOS on standard hardware is left as the only possibility. However, as of now, OctoPOS cannot be run directly on standard hardware.

[1] Hardware support for pull-style DMA transfers would simplify the process at this point.

[2] Strictly speaking [18, §13.3.3], an `at` statement does not start a new activity. Rather, it transports the current activity to another place.

While a version of OctoPOS exists that runs on top of Linux for development and testing purposes, it is clearly unsuited for this comparison. OctoPOS is currently being ported to AMD64 NUMA systems, which will enable a more comprehensive evaluation.

Hence, in the context of this paper, we restrict ourselves to comparing X10 programs with semantically equivalent C programs, where all programs use OctoPOS and run on our many-core architecture. Therefore, our goal is to show that operations regarding activities are cheap in terms of absolute numbers, as well as that the overhead of our X10 runtime system is low compared to directly using the C interface.

Setup. We conducted all running time measurements on an FPGA prototype implementation of our many-core architecture. The prototype consists of 4 tiles with 4 cores each. All cores are unmodified Gaisler SPARC V8 LEON 3 [1, 20] processors. The processors run at 25 MHz and are configured with a 16 KiB instruction cache and an 8 KiB per-core L1 data cache. Additionally, the 4 cores of each tile share a 64 KiB 4-way L2 cache. The tiles are connected by a network-on-chip [9]. The hardware design was synthesized [4] to a CHIPit Platinum system [22], a multi-FPGA platform based on Xilinx Virtex 5 LX 330 FPGAs.

To compile the X10 programs, we use a modified X10 compiler [5] and the custom X10 runtime implementation presented. We use GCC 4.4.2 from the official SPARC toolchain from Gaisler [2] to compile the C programs.

Measurements. We measured all operations with cycle accuracy using a per-tile cycle counter built into our hardware platform. These counters are consistent across tile (or place) boundaries because all counters start running at initialization time of the FPGA boards. Additionally, there is currently no frequency scaling, so all cores on all tiles run with the same frequency of 25 MHz. Hence, not only the running times of local operations but also those of remote operations (such as using `at` in X10) can be measured in cycles.

We repeated each test 20 times and report the minimum running time. Except for the first run with cold caches, all other running times were very similar, so we omit giving standard deviations.

```
// Start local activity
val before = Util.getTimestamp();
async {
  val after = Util.getTimestamp();
  Console.OUT.println("After: " + after);
}
Console.OUT.println("Before: " + before);

// Start remote activity
val other = Place(1);
val before = Util.getTimestamp();
at (other) async {
  val after = Util.getTimestamp();
  Console.OUT.println("After: " + after);
}
Console.OUT.println("Before: " + before);
```

Figure 3. Excerpt from the X10 benchmark program.

First, we look at the running time of two very basic operations: executing a local activity and executing an (asynchronous) remote activity. In X10, this corresponds to the operations `async` and `at async` as shown in the program from Figure 3. Note that neither of the activities accesses any outer variables.

Table 1 gives the running times (in cycles) for both operations. We observe that the absolute number of cycles required for spawning a local activity using `async` is low, taking into account that spawning an activity is a relatively high-level operation that requires system interaction. Additionally, we see that spawning a remote activity using `at async` is not much more expensive than spawning a local activity. Note that the numbers include the time for dispatching the respective *i*-lets. Hence, the running times give the number of cycles it takes until the new X10 activity is actually executing code. If we only take into account the cycles spent until the original activity continues execution, we get 588 cycles (for `async`) and 994 cycles (for `at async`).

	Runtime	
Operation	C	X10
`async`	539	1469
`at async`	1133	1981

Table 1. Delay (in clock cycles) for locally and remotely starting an activity or *i*-let, respectively.

Compared to the direct usage of the C interfaces, the X10 program exhibits slowdowns of roughly factor 3 and factor 2 for spawning a local activity and a remote activity, respectively. This is due to still remaining overhead in the X10 runtime system. For example, each `async` block is converted to a closure object by the compiler. This object must be allocated on the heap via the garbage collector, which is a costly operation in this context. Additional runtime bookkeeping, e.g. for registering the activity at the surrounding `finish` block, as well as procedure-call overhead[3] attribute for the remaining running-time difference.

```
// At statement
val data = new Rail[Byte](size);
val before = Util.getTimestamp();
at (other) {
  val capture = data;
  val after = Util.getTimestamp();
  Console.OUT.println(after);
}
Console.OUT.println(before);

// At expression
val data = new Rail[Byte](size);
val before = Util.getTimestamp();
val dataCopy = at (other) data;
val after = Util.getTimestamp();
Console.OUT.println(after - before);
```

Figure 4. Excerpt from the X10 benchmark program.

Next, we analyze the running times of `at` statements and expressions that access outer variables. We access a one-dimensional array and vary the size of the array from 2^4 to 2^{14}, doubling the size every time. Figure 4 shows an excerpt from the used X10 benchmark program. Accessing `data` inside the `at` statement entails a serialization of the array, copying it to the destination place via a DMA transfer, and the deserialization of the array on the destination place. For the `at` expression, a DMA transfer is also necessary

[3] A peculiarity of the SPARC architecture used is that its registers are arranged in register windows. Each call frame occupies one register window. If the maximum number of register windows is reached, the next procedure call triggers a window-overflow trap, and all register windows are written to memory. Similarly, returning from a procedure may trigger a window-underflow trap. Hence, deep call stacks, which are more common in X10 than in C, are more likely to trigger this behavior.

for returning the result. This corresponds directly to the descriptions in the previous section.

Array size	`at` statement	`at` expression
2^4	13413	27058
2^5	13609	27442
2^6	13863	28127
2^7	14656	29429
2^8	16032	32189
2^9	19017	38309
2^{10}	24806	51235
2^{11}	42932	87847
2^{12}	66630	136349
2^{13}	112564	228551
2^{14}	203220	411306

Table 2. Running times (in clock cycles) for the execution of `at` statements and expressions that access arrays of varying sizes (in Bytes) from the source place.

Table 2 shows the resulting running times (in cycles) for different array sizes. As expected, the running time for small arrays is dominated by the runtime overhead for setting up the inter-place communication. As the array size increases, the running time is dominated by the time needed for serialization and actual data transfer. Hence, X10 applications that frequently transfer small messages would mainly benefit from improvements to the runtime implementation as described in Section 4, such as switching to pull-style DMA transfers, whereas applications that mostly transfer large chunks of data are limited by the interconnect bandwidth.

6. Related Work

Clearly, our work is related to previous work on the work-stealing user-level scheduler of the X10 runtime system [13, 19, 23]. However, we eliminate the user-level scheduler entirely and perform a 1-to-1 mapping of X10 activities to *i*-lets. This becomes possible by making *i*-lets very lightweight and by using cooperative scheduling. This can be viewed as pulling a user-level-like scheduler and its properties down into the kernel.

Chapman et al. [7] report on their experiences using X10 on the Intel Single-Chip Cloud Computer (SCC), which has a tiled architecture similar to our hardware platform. However, they do not use a custom operating system, but instead run a Linux kernel, which implements a traditional process model with preemption.

There are several approaches similar to X10 for parallel programming with fine-grained activities. Cilk and its successors [8, 14] extend the C/C++ programming language with the `spawn` and `sync` keywords, which have similar semantics to `async` and `finish`. The Cilk Plus runtime system dispatches lightweight user-level fibers on top of regular POSIX kernel-level threads which employ work stealing. In contrast, Intel's Threading Building Blocks (TBB) [17] are implemented as a C++ template library, also performing its own user-level scheduling. Unlike X10, both Cilk and TBB assume a shared memory space, and hence do not support distributed programming on PGAS architectures.

Qthreads [24] is an API for lightweight user-level threads. While its primary focus is on hardware architectures that offer dedicated support for massive multithreading, such as the Cray XMT supercomputer, there is also a Pthread-based implementation of the qthreads API for commodity systems. MassiveThreads [15] is another library that implements user-level threads on top of a regular UNIX system. It intercepts blocking I/O operations, performing a context switch to another user-level thread and issuing a non-blocking system call instead. An extension, MassiveThreads/DM, adds functionality for distributed-memory machines with PGAS features.

Project Runnemede [6, 21, 25] introduces *codelets*, which are similar to *i*-lets and are supported directly by the operating system [11]. Codelets are small self-contained units of computation with run-to-completion semantics assumed by default. Similar to *i*-lets, codelets can still be blocked if need be. In contrast to *i*-lets, codelets are expected (but not required) to work functionally, i.e., to only work locally without leaving state behind and with their output only depending on the input values. Additionally, the communication patterns between codelets are restricted. Codelets are arranged in a codelet graph according to their data dependencies, and act as producers and/or consumers, making them similar to dataflow actors in a dataflow graph. Hence, Runnemede makes parallelism more explicit and gives the runtime system additional optimization opportunities. However, programs must either be written in a codelet style in the first place, or a sophisticated compiler is required that decomposes programs written in traditional programming languages into codelets.

The Barrelfish [3] operating system aims into a similar direction as OctoPOS in that it runs as a *multikernel* – multiple OS instances communicating via message passing, even on systems with a cache-coherent shared-memory architecture. Unlike OctoPOS, however, the Barrelfish kernel implements a traditional, heavyweight threading model.

7. Conclusion & Future Work

We presented a way to implement X10 activity management without relying on a user-space scheduler as part of the X10 runtime. This was achieved by mapping activities directly onto the respective OS mechanisms provided by OctoPOS, an operating system mainly targeted towards highly parallel PGAS-style many-core systems. This not only simplifies the implementation of both the X10 runtime as well as the operating system, but our evaluation of the efficiency by measuring running times of various operations provides promising results.

OctoPOS is currently being ported to AMD64 NUMA systems. This enables us to evaluate our approach on mainstream hardware and compare against common Linux-MPI implementations.

Acknowledgments

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Invasive Computing" (SFB/TR 89). Thanks to Jan Heißwolf and Aurang Zaib for designing and implementing the network-on-chip, to Sebastian Kobbe for implementing the global resource manager, and to Stephanie Friederich and David May for synthesizing the many-core architecture to the FPGA platform. We also thank the reviewers for their valuable comments.

References

[1] Aeroflex Gaisler. LEON 3. `http://www.gaisler.com/leonmain.html`, retrieved on 2015-05-13.

[2] Aeroflex Gaisler. LEON bare-C cross compilation system. `http://www.gaisler.com/index.php/products/operating-systems/bcc`, retrieved on 2015-05-13.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3.

[4] J. Becker, S. Friederich, J. Heisswolf, R. Koenig, and D. May. Hardware prototyping of novel invasive multicore architectures. In *Pro-*

ceedings of the 17th Asia and South Pacific Design Automation Conference, ASP-DAC '12, pages 201–206, Jan 2012.

[5] M. Braun, S. Buchwald, M. Mohr, and A. Zwinkau. An X10 compiler for invasive architectures. Technical Report 9, Karlsruhe Institute of Technology, 2012. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000028112`.

[6] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu. Runnemede: An architecture for ubiquitous high-performance computing. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA '13, pages 198–209, Washington, DC, USA, 2013. IEEE Computer Society.

[7] K. Chapman, A. Hussein, and A. L. Hosking. X10 on the Single-chip Cloud Computer: Porting and preliminary performance. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, pages 7:1–7:8, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0770-3.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4.

[9] J. Heisswolf. *A Scalable and Adaptive Network on Chip for Many-Core Architectures*. PhD thesis, Karlsruhe Institute of Technology (KIT), Nov. 2014.

[10] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hübner, R. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel, V. Lari, and S. Kobbe. Invasive manycore architectures. In *Proceedings of the 17th Asia and South Pacific Design Automation Conference*, ASP-DAC '12, pages 193–200, Jan 2012.

[11] R. Knauerhase, R. Cledat, and J. Teller. For extreme parallelism, your OS is sooooo last-millennium. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2012. USENIX. URL `https://www.usenix.org/conference/hotpar12/extreme-parallelism-your-os-sooooo-last-millennium`.

[12] S. Kobbe, L. Bauer, D. Lohmann, W. Schröder-Preikschat, and J. Henkel. Distrm: Distributed resource management for on-chip many-core systems. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, pages 119–128, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0715-4.

[13] V. Kumar, D. Frampton, S. M. Blackburn, D. Grove, and O. Tardieu. Work-stealing without the baggage. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 297–314, New York, NY, USA, 2012. ACM.

[14] C. E. Leiserson. The Cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-497-3.

[15] J. Nakashima and K. Taura. MassiveThreads: A thread library for high productivity languages. In G. Agha, A. Igarashi, N. Kobayashi, H. Masuhara, S. Matsuoka, E. Shibayama, and K. Taura, editors, *Concurrent Objects and Beyond*, volume 8665 of *Lecture Notes in Computer Science*, pages 222–238. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-44470-2.

[16] B. Oechslein, J. Schedel, J. Kleinöder, L. Bauer, J. Henkel, D. Lohmann, and W. Schröder-Preikschat. OctoPOS: A parallel operating system for invasive computing. In R. McIlroy, J. Sventek, T. Harris, and T. Roscoe, editors, *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures*, volume USB Proceedings of *SFMA '11*, pages 9–14, 2011. URL `https://www4.cs.fau.de/~benjamin/documents/octopos_sfma2011.pdf`.

[17] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. ISBN 978-0-59651-480-8.

[18] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 language specification. Technical report, IBM, March 2015. URL `http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`.

[19] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based global load balancing. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 201–212, New York, NY, USA, 2011. ACM.

[20] SPARC International Inc. The SPARC architecture manual, version 8.

[21] J. Suettlerlein, S. Zuckerman, and G. R. Gao. An implementation of the codelet model. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 633–644, Berlin, Heidelberg, 2013. Springer-Verlag.

[22] Synopsis Inc. CHIPit Platinum Edition and HAPS-600 series ASIC emulation and rapid prototyping system – hardware reference manual.

[23] O. Tardieu, H. Wang, and H. Lin. A work-stealing scheduler for X10's task parallelism with suspension. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 267–276, New York, NY, USA, 2012. ACM.

[24] K. Wheeler, R. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '08, pages 1–8, April 2008.

[25] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a "codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.