Interrupt-Driven Cross-Core Invocation Mechanisms on the Intel SCC

Randolf Rotta, Jana Traue, Thomas Prescher, Jörg Nolte
{rrotta,jtraue,tpresche,jon}@informatik.tu-cottbus.de

***Abstract*—On many-core processors, both operating system kernels and bare metal applications need efficient cross-core coordination and communication. Although explicit shared-memory programming and message passing might provide the best performance, they also limit the system's control over scheduling. In contrast, interrupt-driven cross-core invocations provide universal coordination mechanisms that also enable preemptive operations across cores. This paper surveys cross-core invocation mechanisms and their usability with respect to prevalent coordination scenarios. We integrated some of these mechanisms into a bare-metal environment for the Intel SCC processor and will discuss implementation aspects of the interrupt-driven invocations. In conclusion, such invocation mechanisms provide an expressive platform for future operating systems kernels and bare-metal applications.**

***Index Terms*—many-core, remote invocations, bare-metal**

I. Introduction

On many-core processors, both operating system kernels and bare metal applications need efficient cross-core coordination and communication. On past single-core architectures, communication was focused on inter-process communication (IPC) in combination with fast process switching (e.g. LRPC [1]). However, the availability of many cores causes a shift from time multiplexing to space multiplexing [2]. Thus, process switching becomes rare and inter-core communication becomes the common case. In addition, future architectures face severe energy constraints and, thus, for example cores have to sleep instead of busily waiting for events [3].

Irrespectively, the core services of operating systems and runtime environments for bare-metal parallel applications do not change much. The local (core configuration, dynamic power scaling, virtual memory mappings) and global hardware (system memory, devices) has to be managed and basic communication mechanisms are needed to inform about events, to request operations, and to transfer data [4], [5]. On top of these, process coordination schemes such as mutual exclusion, resource allocation, collection of statistics, resource monitoring, load-balancing job queues, and data consistency management are commonly needed [6].

These mechanisms can be based on very different implementation approaches. Conventional shared memory programming can exploit hardware details directly, but scaleable implementations are complex [7] and assume hardware cache coherence, which is not available on all processors. Furthermore, it spreads synchronization mechanisms inside long running threads through the whole system, which, in effect, elides any fine grained scheduling capabilities.

Shared memory programming can be replaced by message-based approaches and vice versa [8]. This enables distributed operating environments based on message passing such as Galaxy OS [4], the Factored OS [2], and Barrelfish [9], which also demonstrate better scalability and performance on shared memory architectures compared to direct shared memory programming. However, message passing is based on explicit send and receive primitives and to avoid communication deadlocks, non-blocking primitives and progress mechanisms are needed. In order to initiate activities, the destination cores have to anticipate them with matching receive operations. At the same time, such messaging primitives impose a strict distinction between local and remote actions. Thus, message passing alone does not answer all coordination needs.

One key purpose of communication is to initiate activities on other cores, for example to offload functionality to dedicated cores [10] or trigger management activities like TLB shoot downs [9]. *Cross-core invocation* (CCI) mechanisms focus on this aspect in order to address the shortfalls of raw message passing. High-level CCI mechanisms address logical system components (e.g. objects) instead of physical cores. This enables invocations on components regardless of their actual location, which highly simplifies the implementation of component-based architectures. CCIs are received implicitly by the runtime environment and are scheduled for later execution. This is essentially equivalent to event-based systems such as JEDI [11], REFLEX [12], and publish/subscribe frameworks [13]. Acting on invocation only also enhances energy saving: Only the per-core idle loop waits for incoming CCIs and, thus, sleep management can easily be integrated [14].

This paper surveys cross-core invocation mechanisms, their usability, and implementation aspects. For this purpose Section II summarizes prevalent coordination scenarios and Section III compares various CCI mechanisms with respect to their usability. Implementation aspects of CCI runtime environments, in particular the scheduling of invocations are discussed in Section IV. Finally, our experiences with implementing such a bare-metal environment on the experimental Intel SCC processor [15] are presented in Section V.

This paper focuses on basic CCI primitives and cannot discuss how efficient scalable shared data structures can be implemented on top of CCI. For multiple-readers/exclusive-writer data, mechanisms like in MESH [16] can be applied. But more research and algorithm design is necessary with respect to sharing under concurrent write access patterns, e.g. shared queues and stacks [7].

II. Coordination Scenarios

The fundamental cross-core invocation initiates asynchronously a selected action on a selected core, that is, the caller will continue its own execution without waiting for the action to be processed. Such CCIs are asynchronous one-side operations, but can be combined to build more convenient coordination mechanisms as surveyed in the next subsections.

A. CCIs with Continuations

Synchronous CCIs continue the caller's activity after the invoked action finished and, often, also a result should be returned to the caller. Examples are allocation requests to resource management services and function offloading to helper cores. This can be achieved with the basic asynchronous mechanism: The callee finishes with a continuation CCI that forwards or returns the results. The continuation can be a fixed part of the callee but passing a description of the continuation along with the callee invocation is more flexible, which is known as a continuation passing [17].

The fault tolerance can be enhanced through timeouts: The caller creates a timeout invocation that will be carried out by the runtime environment after a specified time. Such timeouts can, for example, trigger runtime exceptions. Either the real continuation or the timeout will be processed first and the continuation will deactivate the timeout invocation. Runtime exceptions can be handled similarly with an alternate continuation. For instance, the X10 language has an exception flow model to forward exceptions to callers [18]. Future variables [19] are a convenient way to express continuation passing and TACO [20] implements these with CCIs: The caller passes a pointer to a future variable along with his invocation, the callee returns his results by using a CCI that writes the results to that future variable, and the future variable will wake up the caller.

B. Collective Invocations

Collective invocations apply the same CCI on a group of cores or components in parallel, for example, to perform a TLB shoot down [9], propagate consistency events [16], or initiate parallel computations. A description of the group and the invocation is needed in order to send the actual CCI to each member of the group. Instead of sending the CCIs to each member sequentially, multicast trees reduce the completion time because the tree nodes help to parallelize the CCI propagation. MPI communicators use multicast trees internally to implement collective operations, however, these are not one-sided operations. X10 provides distributed arrays with collective invocations [18]. The TACO object groups [21] allow to configure the tree topology and provide methods to add objects to the group. The tree can also be used as a distributed description of the group because knowing only the root is sufficient to start a collective invocation.

C. Collective Invocations with Results

Often, it is necessary to continue with an activity after all group members in a collective invocation finished their task. One-sided collective gather and reduction operations are equally useful, for example, to query a group of system components in parallel. This can be achieved by combining collective invocations an continuation passing: The collective invocation propagation also distributes internal continuation CCIs. For a collective gather, this continuation forwards the member's result and its group rank and, for a reduction, the continuation sends the result to the caller and merges them there. In order to continue after completion, the caller's continuation is applied when each result arrived.

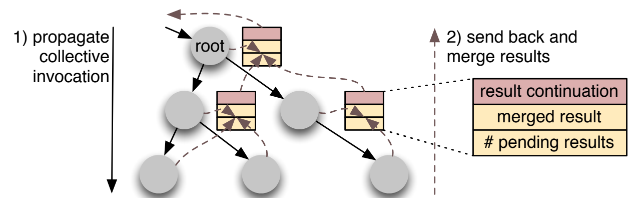

Figure 1. Distributed reduction with temporary helpers. After all results arrived, the result continuation is applied with the combined value and the helper removes itself.

In order to avoid contention, the multicast tree can also be used for distributed result collection as shown in Figure 1. Once all children of a node returned their results, the combined result is sent to the node's parent, which continues until reaching the root. In addition, on shared-memory architectures, the gather operation can write the result into a shared array instead of sending many CCIs to a single destination.

The completion time of collective reductions depends on the communication latency and delays at the multicast nodes Using preemptive CCIs can reduce the propagation delays considerably. For instance, in [22] interrupts were used to speed up the propagation of one-sided data broadcasts. Also, reducing processing overheads during the result collection can reduce the completion time [23].

D. Application Examples

When an application performs a *system call*, conventional systems perform a context switch to kernel mode in order to process the request locally. This is not very useful in a distributed operating system, because the call has to be forwarded to the according service component anyway. Instead, system calls can be implemented by direct CCIs to the service component. The fos operating system [2] uses this approach and demonstrated as good performance as traditional local mechanisms [5]. System calls based on CCIs are also useful in architectures with separate control and compute cores like presented in [24]. There, control engines are specialized cores that manage scheduling and system calls, while all regular CCI actions are carried out on compute cores.

While some implementations provide a single messaging subsystem for small CCIs and large *data transfers* (e.g. MPI, Barrelfish [25]) it can be beneficial to use separate protocols [4], [26], [23]. Such data transfers can be coordinated by CCIs: On shared-memory architectures, the virtual memory

hardware can be exploited to grant read/write access and to transfer ownership of memory pages, which is more efficient when no actual copy is needed [4], [16]. Also, regular copying can be accelerated with shared memory by using free cores to parallelize the copy operation [27].

III. Cross-Core Invocation Mechanisms

In order to express a cross-core invocation it is necessary to select its destination and action. The destination can be a core, an object, or a channel. The invoked action is either selected by the caller (function shipping like in active messages [28] and thread activations [29]) or by the destination (handled by the channel's sink). Optionally, a CCI can be preemptive, that is, processing the requested action is enforced by interrupting the receiver's current activity. These options lead to various CCI mechanisms, which will be discussed in the next subsections.

Throughout all of these mechanisms, anonymous functions emerged as a way to improve flexibility. In C++ these can be implemented without language extensions, for example, in the style of the method-to-functor primitive `m2f` in TACO [20]. This primitive performs argument binding by combining a pointer to a function and argument values into a *functor* (function object). The blocks extension from Apple's Grand Central Dispatch and the delegates in .NET provide equivalent mechanisms. Functors can be passed between cores using messages and can be applied by providing the missing arguments.

A. Remote Procedure Calls (RPC)

Procedure calls are used to transfer control and data in local programs. A caller that requests a service from the callee is suspended until the request is processed. Transferring this widely used abstraction to distributed systems leads to remote procedure calls as CCI mechanism. The following pseudo code gives some examples for such RPCs.

```
1 async(core, procedure);
2 async(core, function, continuation);
3 result = sync(core, function);
```

The caller specifies explicitly the destination core and a functor that represents the desired procedure call with all its arguments (line 1). A continuation can be provided that will process the result on the destination core (line 2). For example, such a continuation could send the result to the caller's core using an RPC like in line 1. The last line shows a synchronous call where the caller is suspended until the result arrived. Transferring the result and waking up to the caller has to be implemented by the runtime environment somehow.

B. Remote Method Invocations (RMIs)

Remote method invocations use global object pointers to select the destination. Such pointers combine the object's location (core) and memory address. In effect, this constructs a partitioned global address space (PGAS) and the object's location determines where the invocation will be carried out.

Unlike the similar RPC mechanisms, the action functor is applied on the destination object, for example, to call methods of that object (line 4 in the example below). This approach provides some flexibility as can be seen with collective invocations (lines 5–7): The same functor is simply applied on each object of the group. For reductions a two-parameter functor describes how the results are merged and the final result is passed as continuation. Similarly, the gather operation can apply the passed continuation on each individual result.

```
4 async(object, method, continuation);
5 ateach(obj group, method);
6 reduce(obj group, method, combiner, continuation);
7 gather(obj group, method, continuation);
```

To combine future variables with RMIs, the continuation functor has a global pointer to the future variable and invokes its assign method using an asynchronous CCI. The future's implementation will suspend and wake up threads that try to read from the future before the value arrived.

C. Invocation Channels

With channels, the caller selects only a destination but the invoked action is selected by each destination individually. The caller writes data to a channel (e.g. emits an event), the channel is connected to sinks on any core, and the sinks invoke an action locally to handle incoming data. Continuation passing is achieved by passing the destination sink for the result as part of the event data.

Such channels provide a different type of flexibility: In contrast to RMIs, the source needs no knowledge about the actual receiver(s) and associated reactions. Sinks can be bound to any action, for example, to a functor that is applied on received data or by assigning a preallocated activity that is scheduled whenever data arrives. Furthermore, sinks can collect and merge incoming data before invoking an action. Future variables are sinks whose actions depend dynamically on the threads that wait for the future's value.

D. Existing Implementations

The Goroutines approach [30] provides an asynchronous invocation primitive with automatic load balancing between cores and channels are used for communication but these do not invoke actions. Inter-processor interrupts (IPIs) provide data-less preemptive channels: Addressing is based on core and interrupt identifiers, and the invoked action is selected by the receiver's interrupt vector. Asynchronous RPC mechanisms are provided by X10's *at* and *async* primitives [18] as well as Chapel's *on* and *cobegin* statements [31].

IV. Runtime Environments

A runtime environment is needed that provides *messaging* for CCI delivery and *scheduling* of pending invocations.

The messaging subsystem has to provide asynchronous *send* primitives and a non-blocking *polling* primitive that receives messages from any source. To ensure consistency, CCIs to the same destination have to be delivered in-order and the combination of polling and scheduling has to ensure in-order processing of CCIs from the same source. Support for small messages of a limited size is sufficient, e.g. a few cache lines. For example, the mailbox system of MetalSVM [26] and protocols from [23] can be used.

Processing an invocation can trigger follow-up tasks. For instance, writing to an event channel's sink usually triggers its event handler, which is executed later. Preemptive invocations require some kind of priority-based scheduling combined with cross-core interrupts to force the receiver to process incoming CCIs and their follow-up tasks in time. Finally, future variables would need the ability to suspend and resume tasks that wait for future values. The next subsection compares basic task types. Then, useful scheduling policies are summarized.

A. Task Types: Activities, Protothreads, and Coroutines

The main difference between the following task types is how their execution context is represented. However, this does not necessarily impact performance [32]. The simplest type are non-blocking *activities*. Once started, an activity runs until completion and, then returns. Hence, activities do not need an individual call stack. Synchronous CCIs such as waiting for future variables is not possible directly, because running activities cannot wait. Instead, result continuations are used to trigger follow-up activities. This style is sufficient, for example, to implement state machines in system components: Events are sent as CCIs to the component and the invoked actions perform state changes and can emit further events.

Protothreads [33] extend activities by a fixed-size processing state. This allows a protothread to suspend itself by storing the next instruction in the processing state and returning control like any completed activity. The execution is continued on the next activation by jumping to the stored instruction. Because protothreads do not have an individual stack, all longterm data has to be stored in the processing state as well. For example, waiting on future variables is achieved by attaching the protothread to the variable. Once the value arrived, the variable reschedules that protothread to continue its execution. Although blocking is still not possible inside nested calls because of the fixed-size processing state, protothread simplify the implementation of complex control flows by converting them into a state machine implicitly.

In contrast, each *coroutine* has an own stack and is started by switching into that stack. Blocking operations suspend the coroutine by storing the execution state on the stack and switching to the next coroutine's stack or back to the scheduler. Thus, a coroutine can suspend inside any nested call.

Interrupting short running activities and protothreads is usually not beneficial. Nonetheless, interrupts can be used to provide a limited number of preemption levels by executing activities of higher priority on the same stack and then continuing the interrupted activity [34]. Coroutines can be suspended from the outside with interrupts by switching out of the coroutine's stack from within the interrupt handler. When the coroutine is resumed at a later time, the interrupt handler is continued, which in turn returns to the coroutine.

Because operations that are used by activities and protothreads must never block, sending CCIs over the network poses a new challenge. In case the CCI message could not be sent because of contention, the send operation has to fail. Coroutines and protothreads can suspend themselves and retry later. Special care is necessary with activities, for example, by repeating the whole activity later or dropping the CCI.

B. Task Scheduling

The task types can be unified as task objects that have a *run()* method and scheduling-specific data such as the task's priority. From the scheduler's point of view, suspended coroutines simply return from their run method. This can be extended to switch memory protection and protection modes when entering and leaving the run method.

When multiple CCIs arrived, an order of execution has to be determined. However, the invocations from the same source should be processed in-order to retain consistency. This can be resolved by restricting invocations to short non-blocking operations and executing them directly from the messaging buffers. These simpler invocations can then invoke longer running actions by enqueueing tasks for the scheduler.

Still, most tasks are short running and, hence, there is no point in scheduling them fairly according to their past processor time as it was common for time-sharing thread schedulers. A simple *First-Come-First-Served* scheduling is often sufficient and faster, because otherwise the scheduling overhead exceeds the task's actual runtime easily. Real-time schedulers assign some kind of priority to each task and process tasks according to the priority, starting with the highest one. The priority can either be determined statically at compile time or dynamically at run-time. An example for the latter is the *Earliest-Deadline-First* scheduling [35], [34].

Another aspect is the integration of application- and system-level scheduling. Some systems separate both strictly and the system's tasks are scheduled only when the application hands over or looses the control flow. For instance, MPI's progress engine is often implemented by a cooperative internal activity scheduler. In contrast, application and system can share a single scheduler, which is necessary for applications with realtime scheduling needs. However, using higher priorities for system tasks would in effect reproduce above separation. An architectural compromise are hierarchical schedulers like the event scheduling tree of the Vortex OS [36].

V. Experiences on the SCC

We implemented a minimal bare-metal environment on the experimental Intel SCC processor [15]. The environment provides interrupt handling, local memory management, various schedulers [34], and one-sided communication based on active messages [23]. Basic CCIs are sent as active messages and are executed directly from the communication memory without any copying. Preemptive CCIs also trigger the external interrupt of the destination core by writing into its on-chip configuration registers. A core has to wait for new CCIs when its task list is empty, which is achieved either by busy polling or by sleeping with the `hlt` instruction until an interrupt signals incoming CCIs. On top of this, REFLEX event channels [12] were implemented, which are global pointers to sinks that trigger a statically allocated activity or protothread.

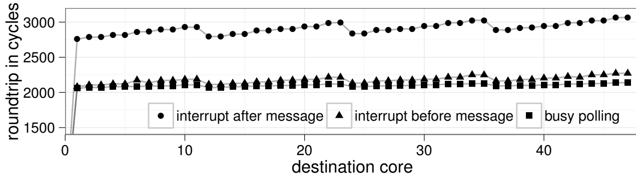

Figure 2. CCI round trip time based on interrupts and on busy polling.

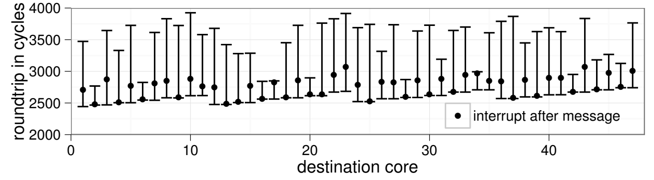

Figure 3. Round trip times before enabling caching of the page tables.

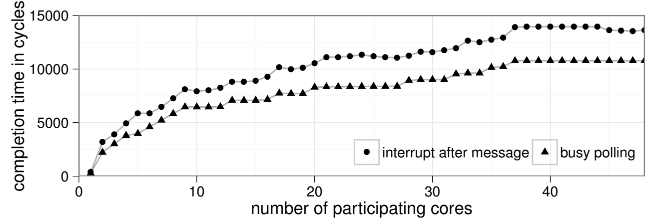

Figure 4. Completion time of collective reduce invocations.

The first experiment studies the impact of interrupt overheads on the CCI roundtrip time: On core 0, an event is written to a channel that points to a sink on another core. There, the sink triggers an activity that emits a result event back to core 0. Figure 2 shows the median round trip times in core clock cycles for the 800x1600x1066 MHz configuration (tile/mesh/memory clock rate) with FIFO scheduling. The round trips took around 2000 cycles with busy polling and around 3000 cycles with `hlt`-based sleeping and sending interrupts after the CCI message. Measuring SCC's `3V3SCC` current meter showed that the processor's power consumption is reduced by a factor of two when all cores are halted instead of busily polling for new messages.

Obviously, the cross-core interrupts for preemption and idle sleeping do not come for free. Performance improvements are possible through latency hiding or overhead avoidance. Latency hiding can be achieved by sending the interrupt before transferring the message because, then, entering the interrupt handling overlaps with writing the message to the communication memory. In the above experiment, this strategy was able to hide the additional interrupt latency nearly completely. Interrupts were sent in both directions by writing to the on-tile configuration registers, which takes 60–90 cycles. Thus, handling a single interrupt takes around 450 cycles. Unfortunately, this strategy could not be used in practice: Sometimes the interrupt handler looks for the message too early. Simply waiting for its arrival is not sufficient because it could have been processed already, but not waiting long enough deadlocks the system inside the `hlt`-based sleeping.

In contrast, overhead avoidance tries to suppress the regular interrupt handling when the system can handle the messages directly. For example, the scheduler can disable interrupts and, instead, check the interrupt status register regularly between executing tasks. The system's reactivity is not degraded as long as these tasks take less time than the interruptions would cost and, thus, interrupts really need to be enabled only for long running tasks. Such overhead avoidance was not effective on the SCC because checking and resetting the interrupt status in the on-chip configuration registers is too costly. This might be resolved by a real processor register that provides fast checking and manipulation of the interrupt status. Also instructions that check for arrived messages more directly than repeatedly fetching data from the communication memory would reduce the polling overhead considerably.

Our first measurements exhibited large variations as shown in Figure 3 with differences up to 1500 cycles between individual round trips. It turned out that some of the TLB misses caused the core to stall because the page tables were mapped as non-cached memory and access to the off-chip memory takes very long when it collides with the DRAM refresh that happens every 64ms. With cached page tables, the jitter shrunk to less than 10 cycles. In consequence, to reduce jitter, caching should be enabled for any mostly-read data in the off-chip memory—including page tables. Sharing parts of page tables with enabled L2 caching is almost possible on the SCC: The collective TLB shoot down invocation just has to evict the modified parts from the L2 caches before invalidating the TLBs. However, the state flags (accessed and dirty bit) are not updated consistently without enabling write-through mode.

On top of the event channels, we implemented collective invocations by using temporary sinks that count and merge incoming results at the multicast tree's nodes. Starting on core 0, one member object per core was added to the group. Figure 4 shows average completion times. A collective reduction with all 48 cores took 14k cycles with interrupts and 11k cycles with busy polling. As expected, the completion time grew logarithmically with the number of participating cores.

VI. CONCLUSIONS

A survey of cross-core invocations at the level of bare-metal applications was presented. With appropriate task scheduling, preemptive cross-core invocations improve the reactivity of the system and enable distributed realtime applications.

We presented experiences gained from implementing such mechanisms on the SCC by using inter-core interrupts. However, the interrupt processing overhead increased the communication latency considerably. Overlapping message transfer and interrupt processing can hide this latency but timing issues pose a challenge while the computational overhead is not reduced at all. Instead, new hardware mechanisms could make preemptive cross-core invocations more efficient and predictable.

ACKNOWLEDGMENTS

We thank Intel for the access to the SCC and the possibility to take part in the MARC program.

REFERENCES

[1] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy, "Lightweight remote procedure call," *ACM Trans. Comput. Syst.*, vol. 8, pp. 37–55, February 1990.

[2] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 76–85, Apr. 2009.

[3] M. B. Taylor, "Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1131–1136.

[4] P. K. Sinha, M. Maekawa, K. Shimizu, X. Jia, H. Ashihara, N. Utsunomiya, K. S. Park, and H. Nakano, "The galaxy distributed operating system," *Computer*, vol. 24, pp. 34–41, 1991.

[5] A. M. Belay, "Message passing in a factored OS," Master's thesis, Massachusetts Institute of Technology, 2011. [Online]. Available: http://hdl.handle.net/1721.1/66407

[6] M. Maekawa, "A classification of process coordination schemes in descriptive power," *International Journal of Parallel Programming*, vol. 9, pp. 383–406, 1980.

[7] N. Shavit, "Data structures in the multicore age," *Commun. ACM*, vol. 54, no. 3, pp. 76–84, Mar. 2011.

[8] H. C. Lauer and R. M. Needham, "On the duality of operating system structures," *SIGOPS Oper. Syst. Rev.*, vol. 13, pp. 3–19, April 1979.

[9] A. Baumann, P. Barham, P. 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*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 29–44.

[10] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," *SIGPLAN Not.*, vol. 44, pp. 253–264, 2009.

[11] G. Cugola, E. Di Nitto, and A. Fuggetta, "Exploiting an event-based infrastructure to develop complex distributed systems," in *Software Engineering, 1998. Proceedings of the 1998 International Conference on*. IEEE, 1998, pp. 261–270.

[12] K. Walther and J. Nolte, "Event-flow and synchronization in single threaded systems," in *First GI/ITG Workshop on Non-Functional Properties of Embedded Systems (NFPES)*, 2006.

[13] P. T. Eugster, R. Guerraoui, and C. H. Damm, "On objects and events," in *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '01. New York, NY, USA: ACM, 2001, pp. 254–269.

[14] A. Sieber, K. Walther, S. Nürnberger, and J. Nolte, "Implicit sleep mode determination in power management of event-driven deeply embedded systems," in *Wired/Wireless Internet Communications*, ser. LNCS. Springer, 2009, vol. 5546, pp. 13–23.

[15] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom *et al.*, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*. IEEE, 2010, pp. 108–109.

[16] T. Prescher, R. Rotta, and J. Nolte, "Flexible sharing and replication mechanisms for hybrid memory architectures," in *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium. Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam*, vol. 55, 2012, pp. 67–72.

[17] G. L. Steele, Jr., "Debunking the 'expensive procedure call' myth or, procedure call implementations considered harmful or, LAMBDA: The ultimate GOTO," in *Proceedings of the 1977 annual conference*, ser. ACM '77. New York, NY, USA: ACM, 1977, pp. 153–162.

[18] 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 *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 519–538.

[19] R. H. Halstead, Jr., "Multilisp: a language for concurrent symbolic computation," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 501–538, October 1985.

[20] J. Nolte, Y. Ishikawa, and M. Sato, "TACO – Prototyping High-Level Object-Oriented Programming Constructs by Means of Template Based Programming Techniques," *ACM Sigplan, Special Section, Intriguing Technology from OOPSLA*, vol. 36, no. 12, December 2001.

[21] J. Nolte, M. Sato, and Y. Ishikawa, "TACO — Exploiting Cluster Networks for High-Level Collective Operations," in *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2001), Brisbane, Australia*. IEEE Computer Society Press, May 2001.

[22] D. Petrović, O. Shahmirzadi, T. Ropars, and A. Schiper, "Asynchronous Broadcast on the Intel SCC using Interrupts," in *Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium*, E. Noulard and S. Vernhes, Eds. Toulouse, France: ONERA, The French Aerospace Lab, Jul. 2012, pp. 24–29.

[23] R. Rotta, T. Prescher, J. Traue, and J. Nolte, "In-memory communication mechanisms for many-cores – experiences with the Intel SCC," in *TACC-Intel Highly Parallel Computing Symposium (TI-HPCS)*, 2012.

[24] R. Knauerhase, R. Cledat, and J. Teller, "For extreme parallelism, your os is sooooo last-millennium," in *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, ser. HotPar'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 3–3.

[25] S. Peter, T. Roscoe, and A. Baumann, "Barrelfish on the intel single-chip cloud computer, barrelfish technical note 005," Tech. Rep., 2010. [Online]. Available: http://www.barrelfish.org/

[26] S. Lankes, P. Reble, O. Sinnen, and C. Clauss, "Revisiting shared virtual memory systems for non-coherent memory-coupled cores," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM, 2012, pp. 45–54.

[27] M. W. van Tol, R. Bakker, M. Verstraaten, C. Grelck, and C. R. Jesshope, "Efficient memory copy operations on the 48-core intel scc processor," in *MARC Symposium*, D. Göhringer, M. Hübner, and J. Becker, Eds. KIT Scientific Publishing, Karlsruhe, 2011, pp. 13–18.

[28] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," University of California, Berkeley, CA, Tech. Rep. UCB/CSD 92/675, 1992.

[29] J. d. Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "TiNy Threads: A thread virtual machine for the Cyclops64 cellular architecture," in *Proceedings of IPDPS'05 - Workshop 14 - Volume 15*, ser. IPDPS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 265.2–.

[30] A. Prell and T. Rauber, "Go's Concurrency Constructs on the SCC," in *Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium*, E. Noulard and S. Vernhes, Eds. Toulouse, France: ONERA, The French Aerospace Lab, Jul. 2012, pp. 2–6.

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

[32] R. von Behren, J. Condit, and E. Brewer, "Why events are a bad idea (for high-concurrency servers)," in *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, ser. HOTOS'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 4–4.

[33] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th international conference on Embedded networked sensor systems*, ser. SenSys '06. New York, NY, USA: ACM, 2006, pp. 29–42.

[34] K. Walther, R. Karnapke, A. Sieber, and J. Nolte, "Using preemption in event driven systems with a single stack," in *The Second International Conference on Sensor Technologies and Applications*, Cap Esterel, France, 2008, pp. 384–390.

[35] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[36] A. Kvalnes, R. Renesse, D. Johansen, and A. Arnesen, "Vortex. an event-driven multiprocessor operating system supporting performance isolation," Universitetet i Tromsø, Tech. Rep., 2003. [Online]. Available: http://hdl.handle.net/10037/367