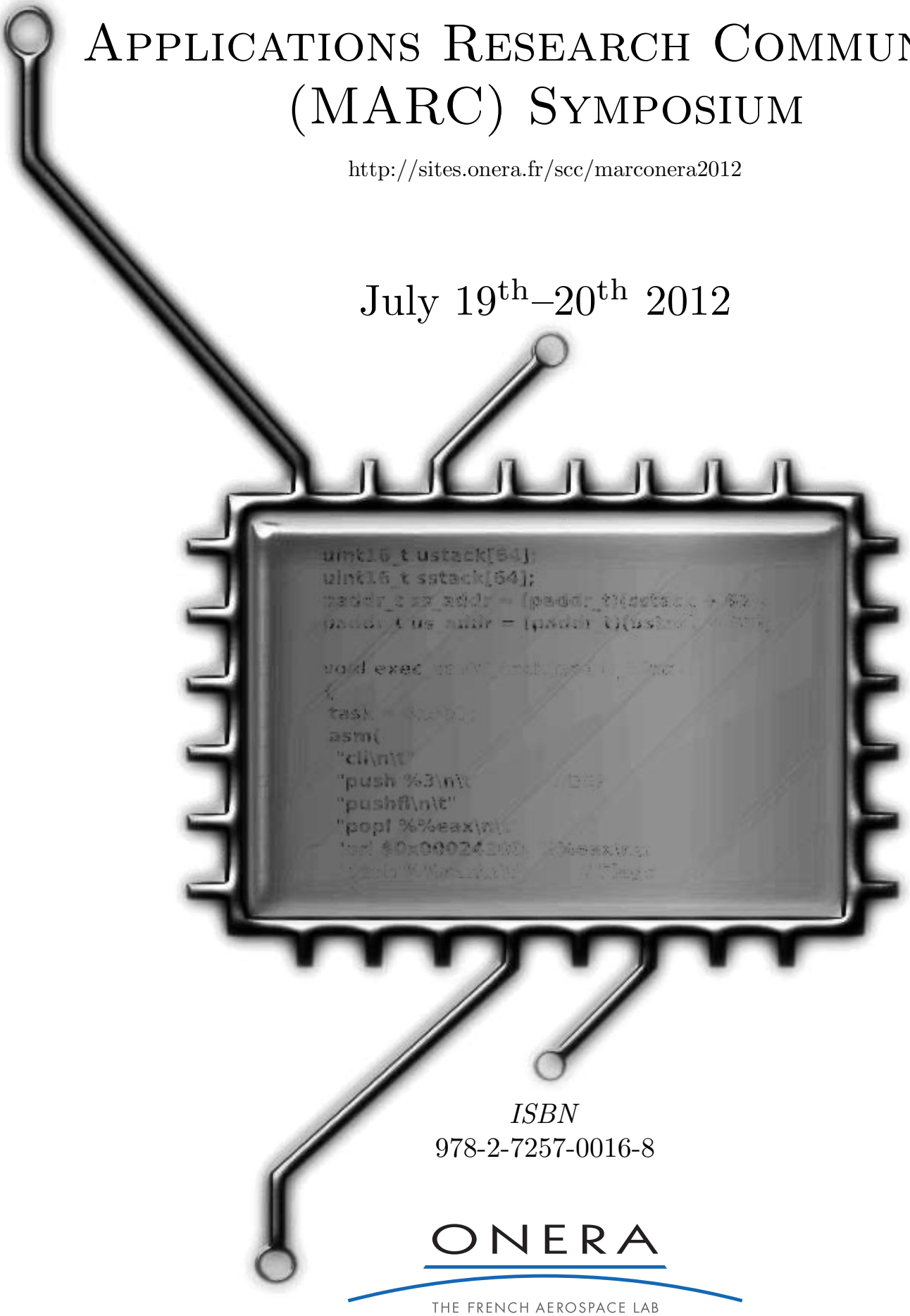
Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium

Eric Noulard

Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium

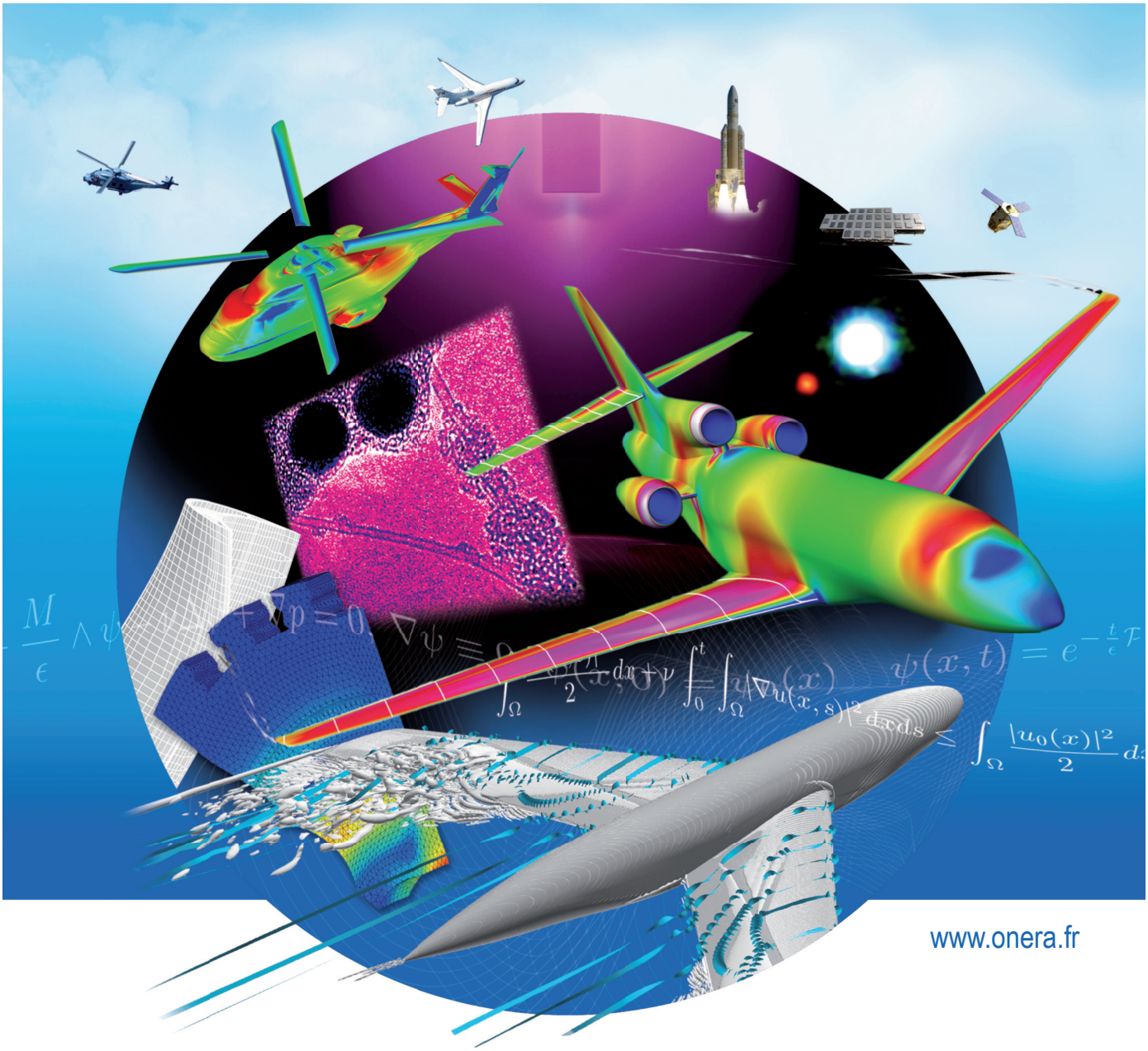
http://sites.onera.fr/scc/marconera2012

July 19th–20th 2012



ISBN
978-2-7257-0016-8

ONERA
THE FRENCH AEROSPACE LAB



www.onera.fr

The leading aerospace and defense research organization in France

To maintain our scientific and technical leadership, ONERA invests in fundamental research and spurs innovation. We are developing the defense technologies that will underpin tomorrow's strategic and tactical systems. We are also working on the enabling technologies for tomorrow's commercial aircraft, to reduce fuel consumption and noise, while further improving safety.

ONERA is a multidisciplinary organization that brings together the talents of 2,000 top experts in energetics, aerodynamics, materials, structures, electromagnetism, optics, instrumentation, atmospheric environment and space physics, complex and onboard systems, information processing and long-term design, Europe's leading center for aeronautical wind tunnels.

Results-oriented, ONERA conducts research in true project mode, based on professionalism, scientific excellence, impartial expertise and confidentiality. Fully two-thirds of our business comes from commercial contracts with firm deadlines, awarded under competitive conditions. Our annual budget is 244 million euros, including 26 million euros to keep our plant and equipment in perfect shape. ONERA works for both governments and industry.

ONERA

THE FRENCH AEROSPACE LAB

return on innovation

 ISBN: 978-2-7257-0016-8

Table of Contents

 ISBN: 978-2-7257-0016-8

MESSAGE FROM THE PROGRAM CHAIR

In continuation of a successful series of events, the 6^{th} symposium of the Many-core Applications Research Community (MARC) took place at ONERA research center in Toulouse. On July 19^{th} and 20^{th} 2012, researchers from different fields presented their current and future work on many-core hardware architectures, their programming models, and the resulting research questions for the upcoming generation of heterogeneous parallel systems.

These proceedings include 12 papers from 4 symposium sessions. Every paper was reviewed by at least three reviewers from the program committee, consisting of:

Pr. Frédéric Boniol	ONERA
Dr. Ulrich Bretthauer	Intel Labs
Dr. Marc Boyer	ONERA
Dr. Carsten Clauss	RWTH Aachen University
Dr. Madeleine Faugère	THALES
Dr. Sylvain Girbal	THALES
Dr. Diana Göhringer	Karlsruhe Institute of Technology (KIT)
Dr. Matthias Gries	Intel Labs
Werner Haas	Intel Labs
Jason M. Howard	Intel Labs
Dr. Michael Hübner	Ruhr University of Bochum (RUB)
Michael Konow	Intel Labs
Dr. Stefan Lankes	RWTH Aachen University
Jean-Claude Laperche	Airbus
Timothy G. Mattson	Intel Labs
Pr. Dr. Jörg Nolte	BTU Cottbus
Dr. Eric Noulard	ONERA
Dr. Claire Pagetti	ONERA
Patrick Pelgrims	ESAT K.U.Leuven
Pr. Dr. Andreas Polze	Hasso-Plattner-Institute
Dr. Philippe Queinnec	IRIT
Michael Riepen	Intel Labs
Dr. Matthieu Roy	LAAS-CNRS
Pr. Pascal Sainrat	IRIT
Pr. Dr. Bettina Schnor	University of Potsdam
Pr. Dr. Theo Ungerer	Universität Augsburg
Dr. Peter Tröger	Hasso-Plattner-Institute
Benoit Triquet	Airbus
Dr. Vincent Vidal	ONERA
Rob F. Van Der Wijngaart	Intel Corp.

We would like to thank our program committee members for their hard work and for their suggestions in the selection of papers. We would like to thank all those who submitted papers for their efforts and for the quality of their submissions. We also would like to thank Nadine Barriety, Marine Bergouts and all members of ONERA/DTIM and ONERA/DCSD for their assistance and support. Special thank to Simon Vernhes for the preparation of the proceedings. Thank you for your active participation in the 6^{th} MARC Symposium.

Eric Noulard,
ONERA, Toulouse Research Center, France.
Toulouse, July 2012.

 ISBN: 978-2-7257-0016-8

Parallelisation of Hard Real-time Applications for Embedded Multi- and Many-cores

Keynote speaker for MARC'ONERA'2012

Prof. Dr. Theo Ungerer
Chair of Systems and Networking
Dept. of Computer Science, University of Augsburg
ungerer@informatik.uni-augsburg.de

Providing higher performance than state-of-the-art embedded processors can deliver today will increase safety, comfort, number and quality of services, while also lowering emissions as well as fuel demands for automotive, avionic and automation applications. Engineers who design hard real-time embedded systems in such embedded domains express a need for several times the performance available today while keeping safety as major criterion. A breakthrough in performance is expected by parallelising hard real-time applications and running them on an embedded multi-core processor, which enables combining the requirements for high-performance with time-predictable execution.

The talk will present research approaches and results of the parallelisation and WCET (Worst-case Execution Time) analysis of industrial hard real-time applications. It shows how a WCET analysis of the communication and synchronization patterns can be performed and how a WCET speedup can be reached for parallelised programs based on parallel design patterns. Research approaches and results of the EC FP-7 projects MERASA (Multi-Core Execution of Hard Real-Time Applications Supporting Analysability, 2007-2011) and parMERASA (2011-2014) are presented. Both projects target timing analysable systems of parallel hard real-time applications running on a embedded multi-core processor. parMERASA investigates time predictable parallelisation for future embedded many-core systems with up to 64 cores.

 ISBN: 978-2-7257-0016-8

Go's Concurrency Constructs on the SCC

Andreas Prell and Thomas Rauber
Department of Computer Science
University of Bayreuth, Germany
{andreas.prell,thomas.rauber}@uni-bayreuth.de

***Abstract*—We present an implementation of goroutines and channels on the SCC. Goroutines and channels are the building blocks for writing concurrent programs in the Go programming language. Both Go and the SCC share the same basic idea—the use of messages for communication and synchronization. Our implementation of goroutines on top of tasks reuses existing runtime support for scheduling and load balancing. Channels, which permit goroutines to communicate by sending and receiving messages, can be implemented efficiently using the on-die message passing buffers. We demonstrate the use of goroutines and channels with a parallel genetic algorithm that can utilize all cores of the SCC.**

I. INTRODUCTION

Go is a general-purpose programming language intended for systems programming [1]. We leave out a general description of Go, and rather focus on its support for concurrent programming, which is not the usual "threads and locks", even if threads and locks are still used under the covers. Programmers are encouraged to "share memory by communicating", instead of to "communicate by sharing memory". This style of programming is reminiscent of message passing, where messages are used to exchange data between and coordinate execution of concurrently executing processes. Instead of using locks to guard access to shared data, programmers are encouraged to pass around references and thereby transfer ownership so that only one thread is allowed to access the data at any one time.

Go's way of thinking is also useful when programming Intel's Single-Chip Cloud Computer (SCC) research processor. The SCC is intended to foster manycore software research, using a platform that's more like a "cluster-on-a-chip" than a traditional shared-memory multiprocessor. As such, the SCC is tuned for message passing rather than for "threads and locks". Or as Intel Fellow Jim Held commented on the lack of atomic operations: "In SCC we imagined messaging instead of shared memory or shared memory access coordinated by messages. [...] Use a message to synchronize, not a memory location." [2], [3] So, we think it's reasonable to ask, "Isn't Go's concurrency model a perfect fit for a processor like the SCC?" To find out, we start by implementing the necessary runtime support on the SCC.

II. CONCURRENCY IN THE GO PROGRAMMING LANGUAGE

Go's approach to concurrency was inspired by previous languages that came before it, namely Newsqueak, Alef, and Limbo. All these languages have in common that they built on Hoare's Communicating Sequential Processes (CSP), a formal language for writing concurrent programs [4]. CSP introduced the concept of channels for interprocess communication (not in the original paper but in a later book on CSP, also by Hoare [5]). Channels in CSP are synchronous, meaning that sender and receiver synchronize at the point of message exchange. Channels thus serve the dual purpose of communication *and* synchronization. Synchronous or unbuffered channels are still the default in Go (when no buffer size is specified), although the implementation has evolved quite a bit from the original formulation and also allows asynchronous (non-synchronizing) operations on channels.

Go's support for concurrent programming is based on two fundamental constructs, goroutines and channels, which we describe in the following sections.

A. Goroutines

Think of goroutines as lightweight threads that run concurrently with other goroutines as well as the calling code. Whether goroutines run in separate OS threads or whether they are multiplexed onto OS threads is an implementation detail and something the user should not have to worry about. A goroutine is started by prefixing a function call or an anonymous function call with the keyword **go**. The language specification says: "A **go** statement starts the execution of a function or method call as an independent concurrent thread of control, or goroutine, within the same address space." [6] In other words, a **go** statement marks an asynchronous function call that creates a goroutine and returns without waiting for the goroutine to complete. So, from the point of view of the programmer, goroutines are a way to specify concurrently executing activities; whether they are allocated to run in parallel is determined by the system.

B. Channels

In a broader sense, channels are used for interprocess communication. Processes can send or receive messages over channels or synchronize execution using blocking operations. In Go, "a channel provides a mechanism for two concurrently executing functions to synchronize execution and communicate by passing a value of a specified element type." [6] Go provides both unbuffered and buffered channels. Channels are first-class objects (a distinguishing feature of the Go branch of languages, starting with Newsqueak): they can be stored in variables, passed as arguments to functions, returned from functions, and sent themselves over channels. Channels are also typed, allowing the type system to catch programming errors, like trying to send a pointer over a channel for integers.

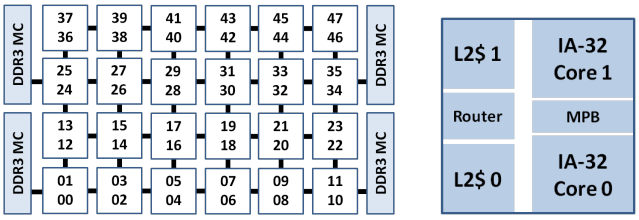


Fig. 1. The 48-core SCC processor: 6x4 tile array (left), 2-core tile (right)

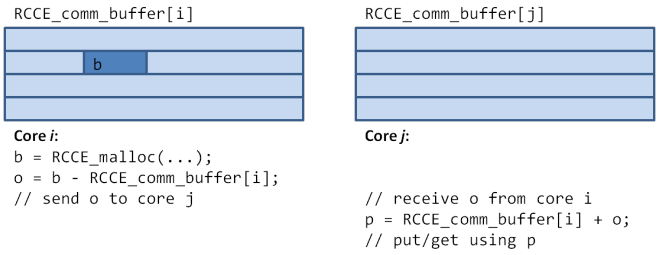


Fig. 2. Accessing buffers in remote MPBs without requiring collective allocations. Because buffer b is not replicated in other cores' MPBs, offset o can't be determined implicitly (as in RCCE_put/RCCE_get), but must be passed between cores.

III. A GLIMPSE OF THE FUTURE? THE SCC PROCESSOR

The Single-Chip Cloud Computer (SCC) is the second processor developed as part of Intel's Tera-scale Computing Research Program, which seeks to explore scalable manycore architectures along with effective programming techniques. At a high level, the SCC is a 48-core processor with a noticeable lack of cache coherence between cores. It does support shared memory, both on-chip and off-chip, but it's entirely the (low-level) programmer's responsibility to avoid working on stale data from the caches—if caching is enabled at all. In the default configuration, most system memory is mapped as private, turning the SCC into a "cluster-on-a-chip", programmed in much the same way as an ordinary cluster.

Message passing between cores is enabled by the inclusion of 16 KB shared SRAM per tile, called Message Passing Buffer (MPB). Programmers can either use MPI or RCCE, a lightweight message passing library tuned to the features of the SCC [7], [8]. RCCE has two layers: a high-level interface, which provides send and receive routines without exposing the underlying communication, and a low-level interface, which allows complete control over the MPBs in the form of one-sided put and get operations—the basic primitives to move data around the chip. RCCE also includes an API to vary voltage and frequency within domains of the SCC, but we won't go into power management issues here.

IV. GO'S CONCURRENCY CONSTRUCTS ON THE SCC

RCCE's low-level interface allows us to manage MPB memory, but with an important restriction. RCCE uses what it calls a "symmetric name space" model of allocation, which was adopted to facilitate message passing. MPB memory is managed through collective calls, meaning that every worker[1] must perform the same allocations/deallocations and in the same order with respect to other allocations/deallocations. Thus, the same buffers exist in every MPB, hence symmetric name space. If we want to make efficient use of channels, we must break with the symmetric name space model to allow every worker to allocate/deallocate MPB memory at any time.

Suppose worker i has allocated a block b from its MPB and wants other workers to access it (see also Figure 2). How can we do this? RCCE tells us the starting address of each worker's portion of MPB memory via the global variable RCCE_comm_buffer. Thus, worker j can access any location in i's MPB by reading from or writing to addresses RCCE_comm_buffer[i] through RCCE_comm_buffer[i] + 8191. Note that in the default usage model, the 16 KB shared SRAM on a tile is equally divided between the two cores. Worker j then needs to know the offset of b within i's MPB. This offset is easily determined on i's side, and after communicating it to worker j, j can get a pointer to b and use this pointer to access whatever is stored at this address. To summarize, any buffer can be described by a pair of integers (ID, offset), which allows us to abandon collective allocations and to use the MPBs more like local stores.

[1]Worker means process or thread in this context. RCCE uses yet another term—unit of execution (UE). On the SCC, we assume one worker per core.

A. Goroutines as Tasks

We have previously implemented a tasking environment to support dynamic task parallelism on the SCC [9]. Specifically, we have implemented runtime systems based on work-sharing and work-stealing to schedule tasks across the cores of the chip. If we map a goroutine to a task, we can leave the scheduling to the runtime system, load balancing included. Scheduling details aside, what **go** func(a,b,c); then does is create a task to run function func using arguments a, b, and c, and enqueue the task for asynchronous execution. Tasks are picked up and executed by worker threads. Every worker thread runs a scheduling loop where it searches for tasks (the details depend on which runtime is used). One thread, which we call the master thread, say, thread 0, is designated to run the main program between the initializing and finalizing calls to the tasking environment. This thread can call goroutines, but it cannot itself schedule goroutines for execution. In addition to the master thread, we need one or more worker threads to be able to run goroutines. This is currently a restriction of our implementation.

Figure 3 shows a pictorial representation of workers running goroutines. Assume we start a program on three cores—say, core 0, core 1, and core 2—there will be a master thread and two worker threads, each running in a separate process. Worker threads run goroutines as coroutines to be able to yield control from one goroutine to another [10]. While a goroutine shares the address space with other goroutines on the same worker, goroutines on different workers also run in different address spaces (sharing memory is possible). This is a deviation from the Go language specification, which states that goroutines run concurrently with other goroutines, *within the same address space*. What we need is a mechanism to allow goroutines

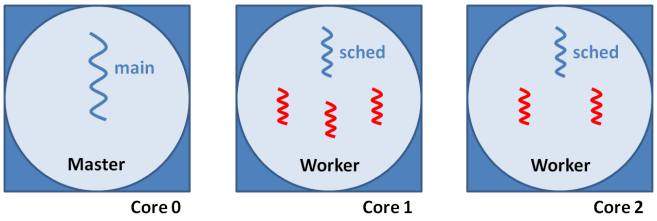


Fig. 3. An example execution of a program with goroutines on three cores of the SCC processor. In this example, worker 1 is running three goroutines, while worker 2 is running two goroutines. The master thread can call goroutines but not run them. Worker threads switch goroutines as needed, whenever a goroutine blocks on a channel.

to communicate, regardless on which core they are running. Channels in shared memory provide such a mechanism.

B. Channels

Our channel implementation takes advantage of the SCC's on-chip memory for inter-core communication. A channel is basically a blocking FIFO queue. Data items are stored in a circular array, which acts as the channel's internal buffer. Channel access must be lock-protected because the SCC lacks atomic operations and only provides one test-and-set register per core for the purpose of mutual exclusion.

A buffered channel of size n has an internal buffer to store n data items (the internal buffer has actually $n+1$ slots to make it easier to distinguish between an empty and a full buffer). If the buffer is full and another item is sent to the channel, the sender blocks until an item has been received from the channel. An unbuffered channel, which is the default in Go when no size is given, is implemented as a buffered channel with an internal buffer to store exactly one data item. Unlike a send to a buffered channel, however, a send to an unbuffered channel blocks the sender until a receive has happened.

A channel (unbuffered) takes up at least 160 bytes—four cache lines to hold the data structure, plus a cache line of internal channel buffer. When we try to write a concurrent program such as the prime sieve presented in the Go language specification [6] and in the Go tutorial [11], we must be aware of channel related memory leaks that can quickly exhaust the MPBs. Go on the other hand, is backed by a garbage collector, which reclaims memory behind the scenes, and which, according to Rob Pike, is in fact "essential to easy concurrent programming" [12]. What the current implementation doesn't include are functions to close a channel and to peek at a set of channels simultaneously (Go's **select** statement, which is like a **switch** statement for channel operations).

C. Channel API

The basic channel API consists of four functions:

Channel *channel_alloc(**size_t** sz, **size_t** n);
Allocates a channel for elements of sz bytes in MPB memory. If the number of elements n is greater than zero, the channel is buffered. Otherwise, if n is zero, the channel is unbuffered. Note that, unlike in Go, channels are untyped. It would be perfectly okay to pass values of different types over a single channel, as long as they fit into sz bytes. Also note that the current implementation does not allow all combinations of sz and n. This is because the underlying allocator (part of RCCE) works with cache line granularity, so we have to make sure that channel buffers occupy multiples of 32 bytes ((n+1)*sz must be a multiple of 32).

void channel_free(Channel *ch);
Frees the MPB memory associated with channel ch.

bool channel_send(Channel *ch, **void** *data, **size_t** sz);
Sends an element of sz bytes at address data to channel ch. The call blocks until the element has been stored in the channel buffer (buffered channel) or until the element has been received from the channel (unbuffered channel).

bool channel_receive(Channel *ch, **void** *data, **size_t** sz);
Receives an element of sz bytes from channel ch. The element is stored at address data. The call blocks if the channel is empty.

Additionally, we find the following functions useful:

int channel_owner(Channel *ch);
Returns the ID of the worker that allocated and thus "owns" channel ch.

bool channel_buffered(Channel *ch);
Returns true if ch points to a buffered channel, otherwise returns false.

bool channel_unbuffered(Channel *ch);
Returns true if ch points to an unbuffered channel, otherwise returns false.

unsigned int channel_peek(Channel *ch);
Returns the number of buffered items in channel ch. When called with an unbuffered channel, a return value of 1 indicates that a sender is blocked on the channel waiting for a receiver.

unsigned int channel_capacity(Channel *ch);
Returns the capacity (buffer size) of channel ch (0 for unbuffered channels).

D. Go Statements

Go statements must be translated into standard C code that interfaces with our runtime library. Listing 1 gives an idea of the translation process. Suppose we start a goroutine running function f(in, out), which operates on two channels in and out. For every function that is started as a goroutine, we generate two wrapper functions, one for creating and enqueuing the goroutine (task), the other for running it after scheduling. Listing 1 shows the corresponding code in slightly abbreviated form. Function go_f creates a task saving all the goroutine's arguments and enqueues the task for asynchronous

 ISBN: 978-2-7257-0016-8

```
void f(Channel *in, Channel *out);

// The go statement
go f(in, out);

// is rewritten into
go_f(in, out);

// with the following definition
// (some details are left out for brevity)
void go_f(Channel *in, Channel *out)
{
  Task task;
  f_task_data *td;

  task.fn = (void (*)(void *)f_task_func;
  td = (f_task_data *)task.data;
  td->in_owner = channel_owner(in);
  td->in_offset = MPB_data_offset(td->in_owner, in);
  td->out_owner = channel_owner(out);
  td->out_offset = MPB_data_offset(td->out_owner, out);

  // Enqueue task
}

// The data structure to hold the goroutine's arguments
typedef struct f_task_data {
  int in_owner, in_offset;
  int out_owner, out_offset;
} f_task_data;

// is passed to the task function that wraps the call to f
void f_task_func(f_task_data *args)
{
  Channel *in, *out;

  in = MPB_data_ptr(args->in_owner, args->in_offset);
  out = MPB_data_ptr(args->out_owner, args->out_offset);
  f(in, out);
}
```

Listing 1: A **go** statement and the translation into tasking code.

execution. Channel references are constructed from channel owner and MPB offset pairs (required to break with the collective allocations model of RCCE, described above), so each channel is internally represented by two integers. The helper functions MPB_data_offset and MPB_data_ptr calculate offsets and pointers based on the MPB starting addresses in RCCE_comm_buffer. The task function f_task_func is called by the runtime when the task is scheduled for execution, after which the goroutine is up and running.

V. EXAMPLE: PARALLEL GENETIC ALGORITHM

To demonstrate the use of goroutines and channels, we have written a parallel genetic algorithm (PGA) that can utilize all the cores of the SCC. We follow the island model and evolve a number of populations in parallel, with occasional migration of individuals between neighboring islands [13].

We choose a simple toy problem: evolving a string from random garbage in the ASCII character range between 32 and 126. More precisely, we want to match the following string that represents a simple "Hello World!" program:

> "#include <stdio.h> int main(void) { printf("Hello SCC!\n"); return 0; }"

The fitness of a string is calculated based on a character by character comparison with the target string, according to $f = \sum_{i=0}^{n}(\text{target}[i] - \text{indiv}[i])^2$, where n is the length of both strings. Thus, higher fitness values correspond to less optimal

```
void evolve(Channel *chan, Channel *prev_in,
  Channel *prev_out, int n, int num_islands)
{
  GA_pop *island;
  Channel *in, *out;

  if (n < num_islands - 1) {
    in = channel_alloc(sizeof(GA_indv), 0);
    out = channel_alloc(sizeof(GA_indv), 0);
    go evolve(chan, in, out, n + 1, num_islands);
  }

  island = GA_create(island_size, target);
  while (GA_evolve(island, migration_rate))
    migrate(island, n, prev_in, prev_out, in, out);
  channel_send(chan, &island->indvs[0], sizeof(GA_indv));
  GA_destroy(island);
}
```

Listing 2: Populations evolve concurrently in goroutines.

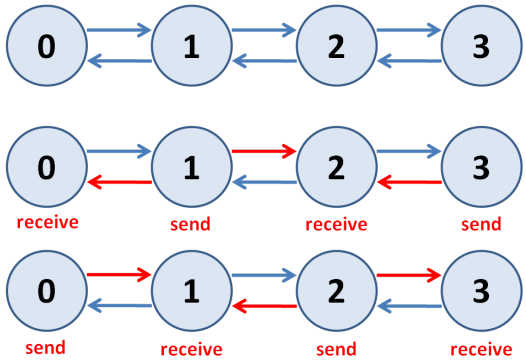


Fig. 4. An example of migration between four islands of a parallel genetic algorithm. Individuals are exchanged over channels in two steps: (1) odd numbered islands send to even numbered neighbors, and (2) even numbered islands send to odd numbered neighbors.

solutions, and our goal is to find the string with the fitness value 0.

Listing 2 shows the code to evolve an island. The GA_* procedures to create, evolve, and destroy a population are not specific to the SCC but part of our generic GA implementation.[2] Islands are created one after another, each in a new goroutine. The main thread of the program starts the evolution by allocating a channel chan, on which the solution will be delivered, and creating the first goroutine with

```
go evolve(chan, NULL, NULL, 0, num_islands);
```

Because the main thread cannot run goroutines, it will block until the evolution has finished when it attempts to receive from chan.

After every migration_rate generations, we migrate two individuals that we pick at random from the current population to neighboring islands. To exchange individuals between two islands a and b, we need two channels: one for sending individuals from a to b, the other vice versa for sending individuals from b to a. Every island other than the first and last has two neighbors and, thus, four channel references to communicate with its neighbors.

[2]The core of the GA uses tournament selection, one-point crossover of selected individuals, and random one-point mutation of offspring individuals.

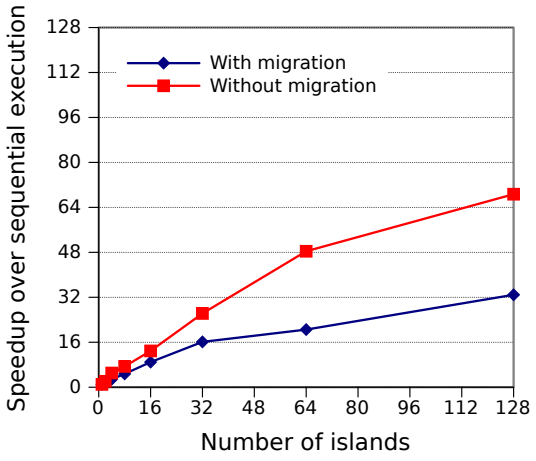


Fig. 5. Speedup with multiple islands over the sequential execution with one island. The total population across all islands has a size of 1280 (island size = 1280 / number of islands). The migration rate is two individuals every ten generations.

Figure 4 illustrates what happens during migration. The example shows four islands numbered from 0 to 3 and the channels between them. Migration is a two-step process. We use unbuffered, synchronizing channels, which require a rendezvous between sender and receiver. First, odd numbered islands send their individuals, while even numbered islands receive in the matching order. Then the process of sending and receiving is reversed.

Figure 5 shows the speedups we have measured for a fixed population size of 1280.[3] Sequential execution refers to the case where we evolve only one island, so everything runs inside a single goroutine. When the number of goroutines exceeds the number of available worker threads, goroutines are multiplexed onto worker threads. Up to the total number of cores (48), we make sure that every new goroutine runs on a separate core, so that goroutines actually run in parallel (though the runtime doesn't allow us to control on which core a goroutine starts execution). Creating more goroutines than we have cores is no problem; the runtime scheduler switches between goroutines whenever active goroutines block on channels during migration.

The toy problem is simple enough that we don't actually need a sophisticated GA that migrates individuals between islands in order to maintain genetic diversity. The same algorithm leaving out migration and instead just switching between goroutines after every migration_rate generations achieves even higher speedups.

VI. CONCLUSION

We have presented an implementation of goroutines and channels, the building blocks for concurrent programs in the Go programming language. Both Go and the SCC share the basic idea of communicating and synchronizing over messages rather than shared memory. The Go slogan "Do not communicate by sharing memory; instead, share memory by communicating" is a good one to keep in mind when programming the SCC. Communication over channels is akin to message passing, but channels are much more flexible in the way they serve to synchronize concurrently executing activities.

Channels can be implemented efficiently using the available hardware support for low-latency messaging. However, problems are likely the small size of the on-chip memory and the small number of test-and-set registers. The size of the MPB (basically 8 KB per core) limits the number of channels that can be used simultaneously, as well as the size and number of data items that can be buffered on-chip. With only 48 test-and-set registers at disposal, allocating many channels can increase false contention because the same test-and-set locks are used for several unrelated channels. As a result, communication latency can suffer. We could support a much larger number of channels in shared off-chip memory, trading off communication latency, but frequent access to shared off-chip DRAM could turn into a bottleneck by itself.

[3]SCC in default configuration: cores running at 533 MHz, mesh and DDR memory at 800 MHz (Tile533_Mesh800_DDR800).

ACKNOWLEDGMENT

We thank Intel for granting us access to the SCC as part of the MARC program. Our work is supported by the Deutsche Forschungsgemeinschaft (DFG).

REFERENCES

[1] "The Go Programming Language," http://golang.org. [Online]. Available: http://golang.org
[2] "Many-core Applications Research Community," http://communities.intel.com/message/113676#113676. [Online]. Available: http://communities.intel.com/message/113676#113676
[3] "Many-core Applications Research Community," http://communities.intel.com/message/115657#115657. [Online]. Available: http://communities.intel.com/message/115657#115657
[4] C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM*, vol. 21, pp. 666–677, August 1978. [Online]. Available: http://doi.acm.org/10.1145/359576.359585
[5] ——, *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.
[6] "The Go Programming Language Specification," http://golang.org/doc/go_spec.html. [Online]. Available: http://golang.org/doc/go_spec.html
[7] T. G. Mattson, R. F. van der Wijngaart, 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 *Proceedings of the 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. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.53
[8] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-weight Communications on Intel's Single-Chip Cloud Computer Processor," *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 73–83, February 2011.
[9] A. Prell and T. Rauber, "Task Parallelism on the SCC," in *Proceedings of the 3rd Many-core Applications Research Community (MARC) Symposium*, ser. MARC 3. KIT Scientific Publishing, 2011, pp. 65–67.
[10] R. S. Engelschall, "Portable Multithreading: The Signal Stack Trick for User-Space Thread Creation," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '00. Berkeley, CA, USA: USENIX Association, 2000, pp. 20–20. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267724.1267744
[11] "A Tutorial for the Go Programming Language," http://golang.org/doc/go_tutorial.html. [Online]. Available: http://golang.org/doc/go_tutorial.html
[12] "Go Emerging Languages Conference Talk," http://www.oscon.com/oscon2010/public/schedule/detail/15299, July 2010. [Online]. Available: http://www.oscon.com/oscon2010/public/schedule/detail/15299
[13] E. Cantú-Paz, "A Survey of Parallel Genetic Algorithms," *Calculateurs Paralleles, Reseaux et Systems Repartis*, vol. 10, 1998.

 ISBN: 978-2-7257-0016-8

Performance of RDF Query Processing on the Intel SCC

Vasil Slavov, Praveen Rao, Dinesh Barenkala, and Srivenu Paturi

***Abstract*—Chip makers are envisioning hundreds of cores in future processors for throughput oriented computing. These processors, called manycore processors, require new architectural innovations for scaling to a large number of cores as compared with today's multicore processors. We report an early study on the performance of RDF query processing on a manycore processor. In our study, we use the Intel SCC, an experimental manycore processor from Intel Labs. This processor has new architectural features, namely, 48 Pentium cores, a high speed, on-chip mesh network to communicate between cores and access memory controllers, on-chip message passing buffers for high speed message passing, and software controlled fine-grained power management. We classify queries based on their I/O footprint and study the impact of two standard models, namely, task and data parallel programming models. Based on our experiments with synthetic and real RDF datasets on the SCC, we conclude that the task parallelism model provides an immediate way to boost the performance of RDF query processing.**

I. INTRODUCTION

Chip makers are envisioning hundreds of cores in future processors for throughput oriented computing. In throughput oriented computing, we expect abundant parallelism opportunities in the workload, and aim to achieve high throughput using a large number of simple cores, while compromising the latency on individual cores [1]. A processor with such large number of cores is called a manycore processor. The cores may be homogeneous or heterogeneous. New architectural innovations for faster on-chip communication and efficient power management are necessary to scale to a large number of cores as compared with today's multicore processors.

In recent years, a few manycore prototypes have emerged (*e.g.*, 80 core processor called Polaris [2], Larrabee [3], Intel Single-chip Cloud Computer (SCC) [4]). Of particular interest to us is the Intel SCC, an experimental manycore processor from Intel Labs. This processor has new architectural features, namely, 48 Pentium cores, a high speed, on-chip mesh network to communicate between cores and access memory controllers, on-chip message passing buffers for high speed message passing, and software controlled fine-grained power management.

In this work, we attempt to understand the benefits and limitations of the SCC for parallel RDF query processing. RDF (Resource Description Framework) is a popular language for representing data on the Web [5]. It enables the interchange and machine processing of data by considering its semantics. The essence of RDF lies in the notion of representing any fact as *subject*, *predicate*, and *object*. Formally, RDF represents resources as a directed, labeled graph where a pair of adjacent nodes denotes two things and the directed, labeled edge represents their relationship. The source node denotes the subject; the sink node denotes the object; and the edge label is the predicate (or property). This "subject-predicate-object" relationship is commonly referred to as an RDF triple. SPARQL is a popular query language for RDF graphs [6]. Using SPARQL, complex graph pattern queries can be expressed on individual RDF graphs as well as across multiple RDF graphs.

In recent years, the RDF data model has become increasingly important in domain-specific applications and the WWW. Through RDF technologies, one can reason over semantic data, which is highly appealing in domains such as healthcare, defense and intelligence, biopharmaceuticals, and so forth. With the rapidly growing size of RDF datasets (*e.g.*, DBPedia [7], Billion Triples Challenge [8]), there is a pressing need for high performance RDF processing tools. With the emergence of manycore processors, it is natural and timely to ask whether *a manycore processor can boost the performance of RDF query processing* – through parallel processing. To the best of our knowledge, there is no published work in this area. Recent studies on the Intel SCC have focused on low level aspects such as on-chip message passing performance, memory access latency, and power and energy consumption on benchmarks from high performance computing [4], [9].

In our study, we adopt standard task parallel and data parallel programming models for parallel RDF query processing. We categorize RDF queries on real and synthetic RDF datasets into two different query workloads based on their I/O footprint – one with small I/O footprint queries and the other with large I/O footprint queries. We study the effect of inter-query parallelism via the task parallel programming model on these workloads. We also study the effect of intra-query parallelism via the data parallel programming model on these workloads.

The rest of the paper is organized as follows. We present background and related work in Section II; we present the methodology of our study in Section III; we present the empirical findings in Section IV; and we conclude in Section V with a note on future work.

II. PRIOR WORK ON RDF QUERY PROCESSING

Today, there are a number of open-source and commercial tools for storing and querying RDF graphs. These tools either

V. Slavov is with the Department of Computer Science and Electrical Engineering, University of Missouri-Kansas City. E-mail: vgslavov@mail.umkc.edu

P. Rao is with the Department of Computer Science and Electrical Engineering, University of Missouri-Kansas City. E-mail: raopr@umkc.edu

D. Barenkala is with the Department of Computer Science and Electrical Engineering, University of Missouri-Kansas City. E-mail: db985@mail.umkc.edu

S. Paturi is with the Department of Computer Science and Electrical Engineering, University of Missouri-Kansas City. E-mail: sp895@mail.umkc.edu

store and process RDF in main-memory, use an RDBMS, or a native RDF database. The popular approach has been to use relational database systems for storing, indexing, and querying RDF [10], [11], [12], [13], [14], [15]. Abadi *et al.* proposed a vertical partitioning approach and leveraged a column-oriented DBMS for achieving an order of magnitude performance improvement over previous techniques [16].

RDF-3X [17] and Hexastore [18] demonstrated that storing RDF data in a single triples table and building exhaustive indexes on the six permutations of (s, p, o) triples can significantly outperform the vertical partitioning approach [16] and also support a larger class of RDF queries efficiently. Recently, BitMat [19] was proposed to overcome the overhead of large intermediate join results in RDF-3X and Hexastore when queries contain low selectivity triple patterns. (Low selectivity implies large result set size.)

There are some RDF stores that operate in shared-nothing clusters (*e.g.*, YARS2 [20], 4store [21], Clustered TBD [22]) by hashing triples/quadruples and distributing them on different nodes in the cluster. Parallel query processing is performed. The scalability of these approaches has been demonstrated on small sized clusters. Weaver *et al.* [23] have studied RDF query processing on supercomputers. More recently, tools for data intensive computing such as Apache Hadoop and Pig have been used for query processing and analytics over RDF data [24], [25], [26]. These approaches are more suitable for batch processing of queries. A few researchers have focused on parallel RDF reasoning [27], [28]. More recently, Huang *et al.* , developed parallel RDF query processing techniques for large RDF graphs [29].

On the Intel SCC, Vidal *et al.* , studied the parallelization of an AI automated planner using a hash-based distribution of tasks [30]. Petrides *et al.* , studied the performance of relational decision support queries on the SCC [31]. However, none of the previous work has studied the performance impact of parallel RDF query processing on the Intel SCC.

III. OUR METHODOLOGY

In this section, we introduce our methodology for parallel RDF query processing on the SCC. Our first approach is to express inter-query parallelism via the task parallel programming model. Our second approach is to express intra-query parallelism via the data parallel programming model. While we adopt standard techniques for task and data parallelism, these techniques provide good insights into the benefits and limitations of the Intel SCC for RDF query processing. The query workloads we study are I/O bound in nature, unlike prior work on SCC [4], [9], which focused on high performance computing benchmarks. We consider two different types of query workloads: one that has relatively smaller I/O footprint and the other that has relatively larger I/O footprint.

A. Message Passing Interface

We use the popular Message Passing Interface (MPI) for writing parallel programs. MPI contains a standard library of routines for writing portable message-passing based programs. The MPI routines that we used for the task parallel and data parallel programming models are listed in Table I. MPI programs essentially create a collection of processes. MPI_Send and MPI_Recv allow a process to exchange messages with another process (point-to-point communication); MPI_Barrier enables processes to synchronize at certain points during execution; and MPI_Bcast, MPI_Scatter, and MPI_Gatherv are collective communication operations, which allow a process to communicate with a group of other processes.

B. Impact of Granularity

In parallel computing, granularity denotes the ratio between the amount of computation to the amount of communication. In fine-grained parallelism, we break a problem into relatively smaller sized computation tasks and therefore, may require more frequent communication between processors. In coarse-grained parallelism, we break a problem into relatively larger sized computation tasks and therefore reduce the frequency of communication between processors. However, fine-grained parallelism enables better load balancing than coarse-grained parallelism. But it may increase communication cost and synchronization overhead. By design, Intel SCC provides a high speed, on-chip network to enable fast communication between cores. Therefore, we attempt to partition the tasks as fine-grained as possible in our experiments. Because the query workloads we study are I/O bound, we use the I/O footprint to characterize the granularity of a task.

MPI routines	Usage
MPI_Send	Is called when a process wants to send a message in its local buffer to another process
MPI_Recv	Is called when a process wants to receive a message from another process
MPI_Barrier	Is called by a process to enter a barrier
MPI_Bcast	Is called by a process to broadcast the message to all processes in the group
MPI_Scatter	Is called by a process to scatter an array of data items to other processes
MPI_Gatherv	Is called by all processes in the group (one receiver, multiple senders) so that the receiver can collect different sized messages from the senders synchronously

TABLE I
MPI ROUTINES USED

C. Task Parallel Programming Model

We express inter-query parallelism via a straightforward task parallel programming model. Each query is regarded as a task. Our model is as follows. On one core, we run the master and on the other cores, we run workers. Algorithm 1 describes the set of actions performed by the master and workers. Lines 1-1 denote the actions taken by the master. Lines 1-1 denote the actions taken by a worker. The master maintains a single task pool. Once the master and workers have started (as MPI processes), each worker sends a message to the master. The master responds to a worker with a query from the task pool. The worker then executes the query locally on the index. (The index is constructed over the entire dataset and is shared by the workers.) Once completed, the worker

Algorithm 1: Task parallel programming model

```
proc @Master()
Create a query pool from a list of SPARQL queries to process
while query pool is not empty do
    MPI_Recv(workerid)
    Remove a SPARQL query q from the pool
    MPI_Send(workerid, q)
execute MPI_Barrier
end
proc @Worker()
while true do
    MPI_Send(master) to request a query
    q ← MPI_Recv(master)
    if q == EOF then
        break
    else
        Execute q locally using the index
execute MPI_Barrier
end
```

repeats the process of requesting and executing queries from the master until the master informs that there are no more queries to execute. Once the master and workers reach the barrier, the task pool has been completely processed.

D. Data Parallel Programming Model

We express intra-query parallelism via a straightforward data parallel programming model. The task of processing a query on the entire dataset is broken down into subtasks, where each subtask consists of executing the query on a different partition of the dataset. Our model for data parallelism is as follows. First, we partition the underlying RDF graph into smaller graphs. We do this by extracting weakly connected directed subgraphs and applying standard graph partitioning techniques if necessary (*e.g.*, METIS [32]). If graph partitioning is applied, then we aim to minimize the number of cut edges. We replicate the cut edges in the partitions. (We ignore the directionality of the edges in the graph while partitioning and assume each edge has unit weight.) In our approach, we may miss results. While overcoming this is a non-trivial research challenge, our goal here is to test whether using partitioned indexes on multiple cores during query processing can provide good speedup for the best case scenario.

Similar to the task parallelism approach described earlier, on one core we run the master, and on the others we run workers. The master selects a query and broadcasts it to the workers and also provides each worker with a bucket id to use during query processing. Each worker executes the query locally on the data in the specified bucket. The partial results are returned to the master. Collecting the results can be done either by sending multiple messages one at a time to the master or using the collective operation MPI_Gatherv. Algorithm 2 describes the steps involved. The master and the workers reach a barrier before the next query is processed.

IV. PERFORMANCE EVALUATION

We used RDF-3X [17], a state-of-the-art RDF query processing engine in our evaluation. RDF-3X was implemented in C++ and was compiled to run on the SCC using a 32 bit GCC compiler. The SCC cores ran Linux and had a NFS mounted file system where the indexes were stored. We did not modify the memory organization/configuration of the SCC and used the default setting.

Algorithm 2: Data parallel programming model

```
proc @Master()
foreach SPARQL query q do
    Let Bid[ ] denote an array of bucket ids
    MPI_Scatter(Bid[ ]) /* Send a different bucket id to each worker */
    MPI_Bcast(q) /* Send the same query to each worker */
    MergeResults()
end
proc @Worker()
while true do
    p ← MPI_Scatter() /* A worker receives one bucket id */
    q ← MPI_Bcast() /* Every worker receives the same query */
    Execute q locally on the index for bucket p
    MergeResults()
end
proc MergeResults()
if Master then
    Collect results from workers using multiple MPI_Recv or single MPI_Gatherv
else
    Send results to master using multiple MPI_Send or single MPI_Gatherv
execute MPI_Barrier
end
```

We implemented the task and data parallel programming models described in Algorithms 1 and 2 using RCKMPI, a modified MPICH2 for the Intel SCC [33]. RCKMPI uses the message passing buffers (MPBs) in the SCC to allow low latency high bandwidth message passing. The SCC platform was initialized to run with tile frequency of 800 MHz, mesh frequency of 800 MHz, and memory controller frequency of 800 MHz.

A. Dataset and Queries

We used two real datasets, namely, YAGO2 [34] and Uniprot [35]. YAGO2 is a semantic knowledge base derived from Wikipedia, WordNet, and Geonames. Uniprot is a comprehensive resource for protein sequence and annotation data. We also generated a synthetic dataset using the Lehigh University Benchmark (LUBM) [36]. The ontology for this dataset is based on a university domain.

Note that the SCC cores generate 32 bit addresses. RDF-3X leverages memory mapping of index files and therefore, recommends 64 bit processors for indexing and querying large RDF datasets. To cope with the 32 bit addressing on the SCC, we indexed a set of triples in each dataset such that the index size was at most 2GB in size, a limit set by the underlying OS. This ensured that RDF-3X successfully ran the queries on the SCC. For YAGO, we indexed 27,331,797 triples; for Uniprot, we indexed 46,972,851 triples; and for LUBM, we

Query	Dataset	I/O footprint	Type	% CPU	Serial time
QY_1	YAGO	14,756 KB	small	29	4.73 secs
QY_2	YAGO	15,004 KB	small	40	9.23 secs
QY_3	YAGO	22,832 KB	small	29	6.51 secs
QY_4	YAGO	33,492 KB	small	21	9.27 secs
QY_5	YAGO	216,564 KB	large	22	82.65 secs
QY_6	YAGO	272,848 KB	large	30	120.08 secs
QY_7	YAGO	332,944 KB	large	43	218.43 secs
QL_1	LUBM	2,668 KB	small	25	1.4 secs
QL_2	LUBM	3,132 KB	small	35	1.47 secs
QL_3	LUBM	9,804 KB	small	19	3.5 secs
QL_4	LUBM	636,204 KB	large	32	299.99 secs
QL_5	LUBM	673,924 KB	large	29	206.58 secs
QU_1	Uniprot	4,468 KB	small	39	2.08 secs
QU_2	Uniprot	10,344 KB	small	39	6.46 secs
QU_3	Uniprot	48,020 KB	large	31	19.39 secs
QU_4	Uniprot	62,188 KB	large	19	15.48 secs
QU_5	Uniprot	166,808 KB	large	17	43.51 secs

TABLE II
INITIAL EVALUATION OF QUERIES

indexed 35,612,176 triples. (The SPARQL queries used for the experiments are listed in a technical report [37].)

B. Query Workload Classification

The queries used in our evaluation are I/O bound in nature. Using the `iostat` command, we measured the I/O footprint of each query. (We dropped the file system buffer cache before running each query by issuing `echo 3 > /proc/sys/vm/drop_caches`.) Based on the I/O footprint, we classified the queries into two categories, `small` and `large`. Queries that were classified `small` had relatively smaller I/O footprint. Queries that were classified `large` had relatively larger I/O footprint. Table II shows the queries and their classification after running each query serially. (The block size used by the filesystem was 4096 bytes.) In addition, the serial time (on a single core) and the % CPU utilization for each query is shown. Queries that had higher CPU utilization (*e.g.*, QY_7), typically returned more results. Note that internally RDF-3X stores long string literals in a *mapping dictionary*, and uses ids in the indexes. At the end of query processing, it maps back these ids to literals using the dictionary. For queries returning large number of results, this cost of mapping becomes non-negligible [17].

C. Evaluation Metrics

We measured the effectiveness of parallel RDF query processing by computing the *speedup* and *efficiency* as the number of available cores was increased. Suppose T_s is the time taken to execute a workload of SPARQL queries on a single SCC core. Suppose T_p is the time taken to execute the queries in parallel (using either data or task parallel programming models) on n SCC cores. (On n cores, we run one master and $n-1$ workers.) The speedup on n cores is computed by the ratio $\frac{T_s}{T_p}$; the efficiency on n cores is computed by the ratio $\frac{speedup}{n}$.

D. Data Partitioning Approach

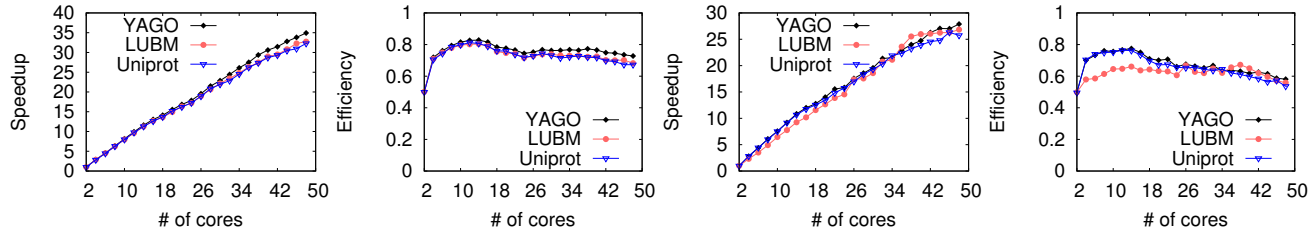
For the data parallel programming model, we partitioned a dataset depending on how many cores were available to run the workers. (Note that partitioning was done once before executing all the queries.) Each worker was assigned one partition and used the index for that partition during query processing. Different approaches were followed for each of the three datasets. The main goal was to assign the triples corresponding to weakly connected directed subgraphs in the RDF graph into buckets. For LUBM, as the generator produced separate RDF files, we grouped the triples from one file and placed it in a bucket. All the files were distributed across the buckets in a round-robin fashion. For Uniprot, we had one single XML/RDF file, and we created fragments of this XML file at points where a new protein was described. The triples from each fragment were stored together in a bucket. All the fragments were distributed across the buckets in a round-robin fashion.

The YAGO2 dataset was available in N-Triples format. First, we extracted graphs of a particular type from the dataset, which we call *star-shaped* graphs. A star-shaped graph is a weakly connected directed graph, where the degree of all vertices except one is exactly 1. All the triples from a star-shaped graph were put into a bucket. On the remaining non-star graphs, we ran the METIS [32] algorithm to partition the graphs. After obtaining n partitions, we assigned the triples for each partition into one bucket. (We replicated the cut edges in each partition.) As mentioned earlier, our approach may miss results.

E. Results

We focus on four possible combinations of workload and parallel programming models, namely, ST (small I/O footprint, task parallelism), LT (large I/O footprint, task parallelism), SD (small I/O footprint, data parallelism), and LD (large I/O footprint, data parallelism). We will refer to these as the ST, LT, SD, and LD models in subsequent discussions. Note that all I/O requests go through the MCPC connected to the SCC platform via the PCIe bus. We measured wall clock time by ensuring a cold cache scenario. (We dropped the file system buffer cache before a query was executed on a core.)

1) The ST Model: The query workload for each dataset consisted of queries marked `small` in Table II. The task pool consisted of these queries put in order and scaled by a factor of 100. (For example, the task pool for YAGO consisted of queries QY_1, QY_2, QY_3, QY_4, ..., QY_1, QY_2, QY_3, QY_4, ...) Figure 1(a) shows the speedup obtained for parallel RDF query processing using Algorithm 1. On 48 cores (1 master + 47 workers), a promising speedup of 34.92, 32.74, and 32.27 was obtained for YAGO, LUBM, and Uniprot, respectively. Figure 1(b) shows the efficiency. For all three datasets, the efficiency reached close to 70% on 48 cores. The tasks were relatively fine-grained due to their small I/O footprints and were well distributed across the workers. There was effective load balancing of tasks across the workers resulting in good speedup and efficiency. (This is evident from the mean and standard deviation of the number of tasks processed by each



(a) Speedup (ST model) (b) Efficiency (ST model) (c) Speedup (LT model) (d) Efficiency (LT model)

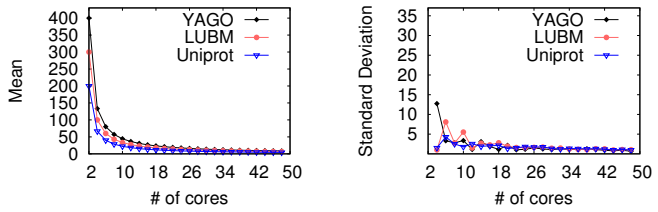
Fig. 1. Results for the task parallel programming model

worker as shown in Figures 2(a) and 2(b).) As shown in Figure 4, the average CPU utilization varied marginally (from 2 to 48 cores), indicating negligible I/O contention in ST.

2) The LT Model: The query workload for each dataset consisted of queries marked `large` in Table II. Similar to ST, the task pool consisted of these queries put in order and scaled by a factor of 33, 50, and 33 for YAGO, LUBM, and Uniprot, respectively. Figures 1(c) and 1(d) show the speedup and efficiency for parallel RDF query processing using Algorithm 1. On 48 cores, the speedup ranged between 25 to 30 for the three datasets. This is promising given that the queries had larger I/O footprint than those used in the ST model. The load was fairly well distributed across the workers. (See Figures 3(a) and 3(b).) As shown in Figure 4, the drop in the average CPU utilization (from 2 to 48 cores) was higher for LUBM and Uniprot as compared to YAGO, indicating higher I/O contention for these datasets.

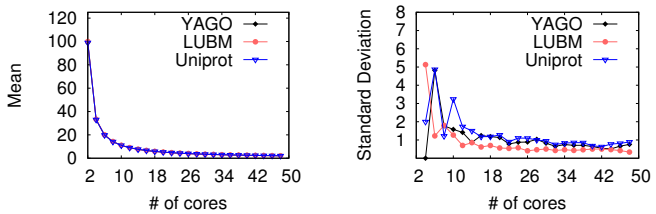
3) The SD Model: The query workload for each dataset consisted of queries marked `small` in Table II. Each query was run multiple times using Algorithm 2. Although the data parallel approach created fine-grained tasks for a query with increasing number of cores, there was load imbalance as many of the workers returned no results on their partitions. This resulted in poor speedup and efficiency as the number of cores was increased. We show the plots in Figures 5(a) and 5(b).

4) The LD Model: The query workload for each dataset consisted of queries marked `large` in Table II. Each query was run multiple times using Algorithm 2. As more cores were used to process a query, I/O contention became an issue. This is evident from the fact that the average CPU utilization for LD was lower than that for LT on all datasets. As a result, poor speedup and efficiency were obtained. We show the plots in Figures 5(c) and 5(d).



(a) Mean (# of tasks per worker) (b) Standard deviation

Fig. 2. Load distribution - ST model



(a) Mean (# of tasks per worker) (b) Standard deviation

Fig. 3. Load distribution - LT model

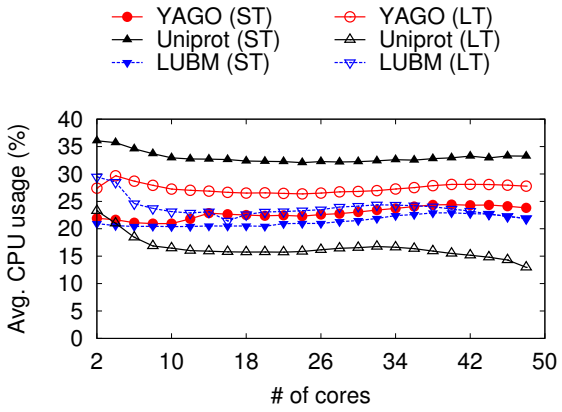


Fig. 4. Average CPU usage with increasing number of cores

F. Summary of Results on the SCC

- The task parallel programming model yielded good speedup and efficiency for parallel RDF query processing. This was true for both small I/O and large I/O footprint queries. The ST model, however, gave better results than the LT model.
- Although the data parallel programming model created fine-grained tasks, the speedup and efficiency for both the SD and LD models were poor due to either load imbalance or I/O contention. Further research is necessary to address these issues.

V. Conclusions and Future Work

We have presented an early study of the performance of parallel RDF query processing on the Intel SCC, an experimental manycore processor. Using real and synthetic RDF datasets, we studied how inter-query parallelism (via

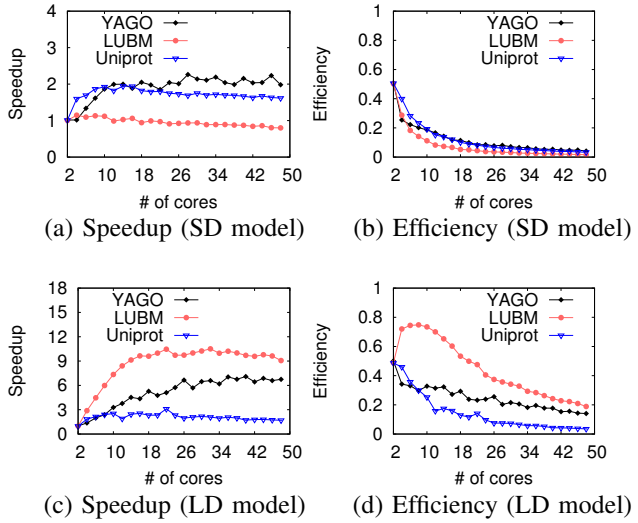


Fig. 5. SD and LD models

the task parallel programming model) and the intra-query parallelism (via the data parallel programming model) affected the performance of RDF query processing. We conclude that the task parallel model provides an immediate way to boost the query processing performance. In the future, we plan to develop new RDF query processing strategies to overcome the challenges posed by the data parallel programming model. We would also like to study the effect of dynamic voltage and frequency scaling of the SCC cores on the performance of RDF query processing.

ACKNOWLEDGEMENTS

We are grateful to the MARC team at Intel Labs for granting us access to the SCC hardware and related software tools. Special thanks to Mark Aughenbaugh and Ted Kubaska for their prompt help. This work was supported in part by a grant from the National Science Foundation (IIS-1115871).

REFERENCES

[1] M. Garland and D. B. Kirk, "Understanding Throughput-Oriented Architectures," *Commun. of ACM*, vol. 53, pp. 58–66, November 2010.
[2] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar, "An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS," in *Solid-State Circuits Conference*, 2007, pp. 98–589.
[3] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Transactions on Graphics*, vol. 27, pp. 18:1–18:15, August 2008.
[4] T. Mattson, R. Van der Wijngaart, 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. of Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, Nov 2010, pp. 1–11.
[5] "Resource Description Framework (RDF)," http://www.w3.org/RDF.
[6] "SPARQL Query Language for RDF," http://www.w3.org/TR/rdf-sparql-query/.
[7] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann, "DBpedia - A crystallization point for the Web of Data," *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 7, no. 3, pp. 154–165, September 2009.
[8] "Semantic Web Challenge," http://challenge.semanticweb.org/.
[9] P. Gschwandtner, T. Fahringer, and R. Prodan, "Performance Analysis and Benchmarking of the Intel SCC," in *Proc. of Intl. Conf. on Cluster Computing*, Sept. 2011, pp. 139–149.
[10] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds, "Efficient RDF Storage and Retrieval in Jena2," in *Proc. of SWDB'03*, 2003, pp. 131–150.
[11] S. Harris and N. Gibbins, "3store: Efficient Bulk RDF Storage," in *Practical and Scalable Semantic Systems*, 2003.
[12] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema," in *Proc. of ISWC '02*, pp. 54–68.
[13] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan, "An Efficient SQL-Based RDF Querying Scheme," in *Proc. of the 31st VLDB Conference*, Trondheim, Norway, 2005, pp. 1216–1227.
[14] L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu, "RStar: an RDF storage and query system for enterprise resource management," in *Proc. of CIKM '04*, Washington, D.C., USA, 2004, pp. 484–491.
[15] J. J. Levandoski and M. F. Mokbel, "RDF Data-Centric Storage," in *Proc. ICWS '09*, Washington, DC, 2009, pp. 911–918.
[16] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable Semantic Web Data Management Using Vertical Partitioning," in *Proc. of the 33rd VLDB Conference*, 2007, pp. 411–422.
[17] T. Neumann and G. Weikum, "The RDF-3X engine for scalable management of RDF data," *The VLDB Journal*, vol. 19, no. 1, pp. 91–113, 2010.
[18] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple Indexing for Semantic Web Data Management," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 1008–1019, 2008.
[19] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, "Matrix "Bit" Loaded: A Scalable Lightweight Join Query Processor for RDF Data," in *Proc. of the 19th Intl. Conference on World Wide Web*, Raleigh, North Carolina, USA, 2010, pp. 41–50.
[20] A. Harth, J. Umbrich, A. Hogan, and S. Decker, "YARS2: A Federated Repository for Querying Graph Structured Data From the Web," in *Proc. of ISWC'07/ASWC'07*, Busan, Korea, 2007, pp. 211–224.
[21] S. Harris, N. Lamb, and N. Shadbolt, "4store: The Design and Implementation of a Clustered RDF Store," in *Proc. of 5th Intl. Workshop on Scalable Semantic Web Knowledge Base Systems*, 2009, pp. 94–109.
[22] A. Owens, A. Seaborne, N. Gibbins, and M. Schraefel, "Clustered TDB: A Clustered Triple Store for Jena," in *Technical Report, Electronics and Computer Science, University of Southampton*, 2008.
[23] J. Weaver and G. T. Williams, "Scalable RDF Query Processing on Clusters and Supercomputers," in *Proc. of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems*, 2009.
[24] P. Castagna, A. Seaborne, and C. Dollin, "A Parallel Processing Framework for RDF Design and Issues," HP Labs, Bristol, Tech. Rep., 2009, www.hpl.hp.com/techreports/2009/HPL-2009-346.pdf.
[25] R. Sridhar, P. Ravindra, and K. Anyanwu, "RAPID: Enabling Scalable Ad-Hoc Analytics on the Semantic Web," in *Proc. of ISWC '09*, 2009, pp. 715–730.
[26] M. F. Husain, J. McGlothlin, M. M. Masud, L. R. Khan, and B. Thuraisingham, "Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, pp. 1312–1327, 2011.
[27] J. Weaver and J. A. Hendler, "Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples," in *Proc. of ISWC '09*, Chantilly, VA, 2009, pp. 682–697.
[28] J. Urbani, S. Kotoulas, E. Oren, and F. Harmelen, "Scalable Distributed Reasoning Using MapReduce," in *Proc. of ISWC '09*, 2009, pp. 634–649.
[29] J. Huang, D. J. Abadi, and K. Ren, "Scalable SPARQL Querying of Large RDF Graphs," *PVLDB*, vol. 4, no. 11, pp. 1123–1134, 2011.
[30] V. Vidal, S. Vernhes, and G. Infantes, "Parallel AI Planning on the SCC," in *4rd Many-core Applications Research Community Symposium (MARC 2011)*, Potsdam, Germany, 2011, pp. 1–6.
[31] P. Petrides, A. Diavastos, and P. Trancoso, "Exploring Database Workloads on Future Clustered Many-Core Architectures," in *3rd Many-core Applications Research Community Symposium (MARC 2011)*, Ettlingen, Germany, 2011, pp. 81–84.
[32] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, pp. 359–392, December 1998.
[33] I. A. C. Urena, "RCKMPI User Manual," *Intel Braunschweig*, 2011.
[34] J. Hoffart, F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, and G. Weikum, "YAGO2: Exploring and Querying World Knowledge in Time, Space, Context, and Many Languages," in *Proc. of WWW '11*, 2011, pp. 229–232.
[35] "Uniprot RDF Distribution," http://www.uniprot.org/downloads.
[36] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, pp. 158–182, October 2005.
[37] V. Slavov, P. Rao, D. Barenkala, and S. Paturi, "Towards RDF Query Processing on the Intel SCC," University of Missouri-Kansas City, Tech. Rep. TR-DB-2012-01, 2012, http://r.web.umkc.edu/raopr/TR-DB-2012-01.pdf.

 ISBN: 978-2-7257-0016-8

Data Sharing Mechanisms for Parallel Graph Algorithms on the Intel SCC

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

***Abstract*—On many-core processors that do not provide hardware cache coherence, using shared memory in parallel computations is challenging. Reverting to pure message passing would avoid consistency issues, but replicating large shared datasets by messages is less efficient than accessing them directly through shared memory. The TACO-MESH framework provides lightweight remote method calls and shared objects with software-managed consistency. This paper presents experience from porting a graph partitioning algorithm to the framework. A performance evaluation on the experimental Intel SCC processor, which has no hardware cache coherence, shows that parallelization can be efficient despite the overhead of software-level consistency management.**

***Index Terms*—many-core, shared memory, cache coherence, graph partitioning**

I. INTRODUCTION

The number of cores in current many-core architectures is increasing while the performance of most cores decreases in favor of smaller cores [1]. Technically, many-cores are hybrids combining aspects of distributed as well as shared memory systems [2], because internal networks connect the cores to exchange messages between them and connect to memory modules that provide direct access to shared memory.

Many architectures employ data and instruction caches distributed over the network to reduce access latencies and memory traffic by exploiting locality. On cache coherent architectures (e.g. Intel MIC [3], and Tilera [4]), the caches implement a coherence protocol in hardware. However, coherence protocols face significant scaling issues compared to message passing [5], [6], [7], [8]. The SCC processor is an experimental *concept vehicle* created by Intel Labs as a platform for many-core software research. All of its cores have private caches, but these do, deliberately, not provide hardware-level coherence [9]. Instead, message passing between cores can be used to implement software-level coherence. At the other extreme end are architectures without caches that instead use large shared on-chip memories and a huge number of simpler cores (e.g. IBM Cyclops64 [10], Adapteva [11], and some stream processors in GPUs).

The SCC and other architectures without hardware cache coherence can be treated like distributed systems to work around this limitation. However, passing large messages inside shared memory systems is inefficient, because composing and receiving large messages evicts large portions of the sender's and receiver's caches. Finally, the message data is just copied from and to main memory and will also compete for space in shared caches (e.g. on Intel MIC). Consequently, parallel programming models and frameworks for many-cores should avoid large unnecessary data copies. Instead, software layers that manage the cache coherence for actual shared data should be integrated into the programming models.

TACO [12] provides a partitioned global address space, remote method invocations, and collective operations; it features a highly efficient messaging backend on the SCC. [13]. On top of that, MESH [14], a framework for memory-efficient sharing, introduces direct access to shared data and a consistency layer for shared objects while using TACO for coordination.

This paper presents experience gained from porting central parts of a complex graph partitioning software for modularity clustering [15] to the SCC. The next section introduces the graph partitioning problem, the local search algorithm, and the employed graph data structures. We combined the graph data structures with the MESH consistency layer and extended MESH with SCC-specific software-level cache coherence, which are described in Section III. Section IV discusses results obtained from micro-benchmarks and the parallelized graph partitioning algorithm. Finally, we discuss related work and provide concluding remarks.

II. MODULARITY GRAPH PARTITIONING

Graph partitioning can be applied in the analysis of social, biological and technical networks and is, in this context, also known as *graph clustering*. For example, persons in social networks can be modelled by graph vertices, edges connect pairs of related persons, and edge weights quantify how often both persons interacted in the past. Graph clustering is searching groups of highly related vertices and, in our particular application, a partitioning of the vertices with dense connections within groups and sparse connections in between.

The *modularity* by Newman and Girvan [16] is a popular quality measure that directs the search for interesting vertex partitions. It is based on the difference between the fraction of *observed* within-group edges and the *expected* fraction. The expected fraction is based on a stochastic model where the end-vertices of edges are chosen at random, and the probability that an end-vertex of an edge attaches to a particular vertex is proportional to the vertex weight [17]. Compared to other clustering quality measures, the modularity is still easy to calculate. Nevertheless, in contrast to the more traditional quality measures for load-balancing, modularity has data dependencies that disallow some well-known performance optimizations. Thus, the modularity is a good representative for a broader class of graph partitioning problems.

 ISBN: 978-2-7257-0016-8

```
1: find best v, D, ΔQ_{v,D} over all v ∈ V
   by collective operation over all workers
2: if ΔQ_{v,D} ≤ 0 then exit loop
   // move vertex v from partition C to D:
3: if D is empty then create new partition D
4: set current partition of v to D
5: update partition weights w(C) and w(D)
6: increment partition size of D
7: decrement partition size of C
8: if size of C is 0 then delete partition C
```

Figure 1. A step of the globally greedy vertex moving algorithm.

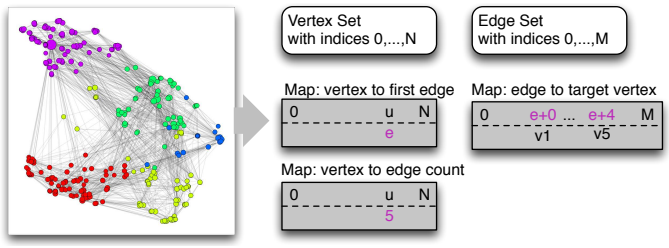


Figure 2. A graph with N vertices, M edges, and three mappings that connect edges and vertices. Highlighted in the mappings is the representation of 5 edges that start in the vertex u and connect to the vertices $v_1 \dots v_5$.

The next subsection introduces a simple parallel graph partitioning algorithm that will be used for the evaluation of the software-level consistency management. The second subsection describes the data structures used by this algorithm.

A. Globally Greedy Vertex Moving

The problem of finding a clustering with maximum modularity for a given graph is NP-hard [18], and existing exact algorithms are usable only up to a few hundred vertices. In practice, modularity is almost exclusively optimized with heuristic algorithms, and experimental results indicate that relatively simple algorithms based on local search can be highly efficient and effective (e.g. [15]).

One of the simplest local search algorithms is *globally greedy vertex moving* as summarized in Figure 1: In each step, the modularity improvement $\Delta Q_{v,D}$ of moving vertex v to partition D is computed for all pairs of vertices and partitions (line 1) and the globally best move is applied by modifying the partitioning data (lines 3–8). This is repeated until the best move does not increase the modularity (line 2). Faster algorithms exist that are similarly effective but, unfortunately, also more complex. We chose the simplest algorithm to focus on the consistency management of the shared data. Our parallel algorithm uses a set of worker cores to parallelize the computation of ΔQ by dividing the vertex set into equally sized subsets. In each step, the workers compute the improvements of their vertices and return v, D, and $\Delta Q_{v,D}$ of their best move. Then, the master worker selects and applies the best of these moves.

Moving a vertex $v \in V$ from its current partition $C \subseteq V$ to another partition $D \subseteq V$ increases the modularity by

$$\begin{aligned} \Delta Q_{v,D} &= 2\frac{f(v,D) - f(v,C\backslash v)}{f(V,V)} \\ &- 2\frac{w(v)w(D) - w(v)w(C\backslash v)}{w(V)^2}, \end{aligned}$$

where $f(A,B)$ is the sum of edge weights between two vertex sets and $w(A)$ is the sum of vertex weights in the vertex set. From an algorithmic perspective, this means that the modularity of a partitioning can be quickly updated after each move without recomputing it from scratch. The partition weights $w(C)$ and $w(D)$ of source and destination partition can be updated after each vertex move by using the weight $w(v)$ of the moved vertex.

Moreover, the search space is restricted, because moving a vertex to a non-adjacent partition ($f(v,D) = 0$) never increases the modularity more than moving it to a new, previously empty partition ($w(v)w(D) = 0$). However, the edge weights $f(v,D)$ must be recomputed in each step because storing and updating them is less efficient. At each vertex v, the algorithm scans over its adjacent vertices u and increments $f(v,\mathcal{C}(u))$ by $f(v,u)$, where $\mathcal{C}(u)$ is the partition containing vertex u. A sparse mapping from partitions to accumulated edge weights is used to store $f(v,\mathcal{C}(u))$ and is initialized with zero weights. Internally, each worker uses an own mapping instance for $f(v,D)$, but all workers share the graph, weights, and the current partitioning. Altogether, finding the globally best move requires a constant time per edge and applying a move costs constant time.

B. Data Structures for Graph Partitioning

The original graph partitioning software [15] had to deal with many different algorithms that stored different data about vertices, edges and partitions internally. To handle this diversity, the concept of *Index Spaces* and *Mappings* was introduced to separate navigation through structures from the algorithm's internal data. In general terms, Index spaces represent a collection of indices and methods to navigate over these indices, while mappings are key-value stores that use the indices as keys. This separation allows algorithms to reuse existing spaces and mappings and to create own mappings for internal data on top of these spaces.

Figure 2 depicts one of many methods to model a static graph data structure. The graph has two spaces, namely a set of vertices and a set of edges, and three mappings to connect vertices and edges. The actual index values of the vertices and edges are irrelevant to the algorithms because they are passed to the mappings transparently and only there the index is used to retrieve the corresponding data value. In case the mappings are implemented with arrays, these will work most efficiently, when the vertices and edges are numbered consecutively beginning from zero.

Our implementation is based on the class `RangeSpace` as fundamental index space. It represents a continuous set of indicies from zero to an upper bound and provides a forward iterator as well as methods to increase the upper bound. The mappings are implemented with arrays. Each time the size of

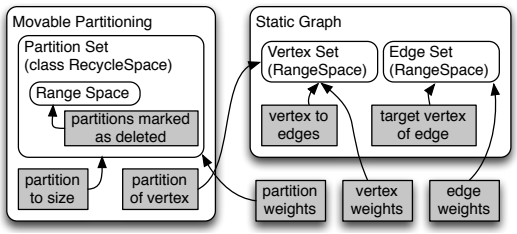


Figure 3. Composition of the graph and partitioning structures. Boxes with round corners are index spaces and all other boxes are mappings. The arrows point to the key space of the mappings.

a space is increased by creating a new index, it is necessary to also resize the arrays of all dependent mappings, which is automated with an observer pattern. In order to minimize the resizing overhead, the maps reserve larger arrays and the space notifies the maps only when the smallest reserve is depleted.

More advanced spaces, for example subsets and graphs, are implemented by using `RangeSpace` and helper mappings. Figure 3 gives an overview of the structures necessary to represent the graphs, vertex partitions, and weights that are used by the graph partitioning algorithm. During the search step, none of these data structures will be modified. To apply the best move, the set of partitions, the mapping from vertices to their current partition, and the partition weights of the source and destination partition will be modified.

III. SOFTWARE-LEVEL CACHE COHERENCE

The MESH framework [14] provides basic facilities for shared memory and shared objects. Its implementation uses the TACO framework [12] to coordinate memory allocation and consistency events. The first subsection describes how we supply MESH with shared memory on the Intel SCC. The second subsection introduces software-level consistency for shared objects and discusses three SCC-specific implementations. Finally, the interactions between shared objects, index spaces, and mappings are discussed in the last subsection.

A. Shared Memory on the Intel SCC

The memory management of MESH is based on separate allocators for shared and core-private physical memory pages and a global allocator for logical pages with aligned addresses over all cores. While most of these allocators are platform independent, a SCC-specific allocator for shared physical memory is necessary. The SCC consists of 32-bit Pentium cores and, thus, each core can address only 4GB of memory. A translation table (LUT) between each core and the on-chip network translates from physical to system addresses that provide a larger address space and contain the network destination (e.g. select one of the four memory controllers). The LUT has 256 entries of 16MB blocks and around 40 entries provide 640MB of private memory for each core.

To acquire direct access to shared memory, it is necessary to make some of each core's private memory accessible to all cores by remapping unused entries in all LUTs. Intel's POPSHM kernel extension provides information about private memory that can be used for this purpose. We tested which LUT entries can be used for the remapping. This approach makes it possible to share 2.5GB of memory between all cores.

The SCC variant of the shared page allocator also provides means to map pages in cached (DCM), non-cached (NCM) and write-through (MPBT+WT) mode. Pages in DCM mode use the core's L1 and L2 caches and require manual coherence management. In contrast, the WT mode uses just in the L1 caches and SCC's write combine buffer to collect writes to a line before sending the modifications to the main memory.

B. Shared Objects with Consistency Management

Special constructors are used to create shared objects. These return a *sharing pointer* that contains the address of the actual object and the address of its *consistency controller* objects. The object is allocated in shared memory but the controllers are allocated in core-private memory at each core. Sharing pointers can be passed between cores using remote method calls or shared memory, because the object resides at the same logical address on all cores and the controller has an instance at the same logical address on each core. Immediate access to shared objects is prohibited. Instead, a temporary *access proxy* has to be created, which triggers consistency events on construction and destruction. The *reader proxies* provide access only to non-modifying (`const`) methods of the shared object, while *writer proxies* allow access to all methods. The consistency events are methods of the consistency controller.

For cache coherence on the SCC it is necessary to invalidate stale data in the caches manually when acquiring read or write access. To ensure that modified data is written back to the main memory before other cores read it, a cache write-back is necessary when releasing write access. Obviously, write-back and invalidation is necessary only when the shared object was actually modified. The next paragraphs present three approaches to implement this coherence management. They support multiple concurrent readers, but only a single non-concurrent writer. Thus, applications have to ensure this concurrency restriction on their own. The structure of our graph partitioning algorithm already guarantees this.

The *Broadcast* (BC) controller has a *needs-flush* flag on each core. When acquiring access to its shared object and the flag is set, the object's memory range is invalidated in the cache. When releasing write access, the cache is flushed to write back all modifications to the main memory and a TACO collective operation sets *needs-flush* on all other cores. To save time the flushing and the collective operation overlap.

In case flushing is much faster than the broadcast, some time can be saved by starting the broadcast earlier. The *Overlapped Broadcast* (OV) controller achieves this by initiating the broadcast already when write access is acquired. As a side-effect, the broadcasts of several modified shared objects can overlap. However, with this approach additional state data for the pending broadcast has to be stored in the consistency controller and the controller could not skip the broadcast if the object was not actually modified.

A completely different invalidation mechanism similar to [19] is used by the *Timestamp* (TS) controller. Each shared object has a timestamp (generation counter) that is stored in the on-chip SRAM of the SCC and the consistency controller on each core has a copy of the last seen value. When releasing write access, the object is flushed to write back all modifications and the object's counter is incremented.

C. Sharing Index Spaces and Mappings

The graph data is implemented with index spaces and mappings (c.f. Section II-B). However, allocating `RangeSpace` as a shared object is not sufficient because it contains a resizable array of pointers to observers (the dependent mappings). This array is invisible to the consistency controller and would be missed when flushing the caches partially. To solve this, shared objects can inherit from the special class `SharingAware` that instructs the consistency controller to also call type-specific consistency management methods. `RangeSpace` uses these to flush the internal observer array.

The mappings use internal data arrays similar to `RangeSpace`. They are managed using `SharingAware`. Furthermore, all mappings have to register themselves as observer at their key space by providing a sharing pointer to themselves. Because a normal shared object does not know its own consistency controller, the `SharingAware` base provides a method to get such a sharing pointer. SCC's small 32-bit address space makes it expensive or difficult to increase the size of arrays. To overcome this, our mapping implementations split the data array into 4kB chunks and use a small lookup array to address these chunks. This allows to increase the maps by adding chunks without moving data.

Three variants of mappings were implemented: The `ChunkedMapDCM` uses cached shared memory for all chunks and the lookup table. The overhead of flushing the relatively large data chunks is avoided by the `ChunkedMapNCM` variant, which still caches the lookup table but not the data chunks. The third variant, `ChunkedMapWT`, caches the data chunks only on the L1 cache using SCC's MPBT+WT memory mode. Any communication by message passing will automatically invalidate cached chunk data and force the write combine buffer to write the modified data back to the main memory. The latter two implementations specialize the writer access proxy to allow write access to the mapping data without triggering unnecessary consistency events.

Composition, that is shared objects using other shared objects, yields an interesting situation: Creating a usual access proxy to the composed object and then calling its methods will be inefficient because each call would have to acquire and release access to the other shared objects, triggering a multitude of consistency management events. To avoid this, we specialize the access proxies of such objects and move the methods from the object into the access proxy. For example, the graph reader access proxy provides the usual methods to navigate through the graph, but it acquires read access to all necessary components just once when the proxy is constructed.

Table I
ACCESS LATENCIES TO SCC'S MEMORY (IN CYCLES).

	on-chip SRAM		off-chip DRAM		
	NCM	MPBT	NCM	MPBT+WT	DCM
w	5	29	11	35	98
g	51–84	78–108	106–137	130–161	107–123
r	58–91	67–97	117–145	123–153	150–174

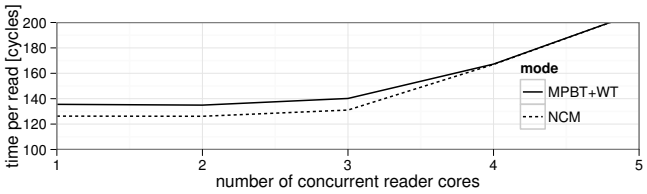


Figure 4. Approximate read overhead as observed by a core with concurrent read accesses to the same controller.

IV. PERFORMANCE EVALUATION ON THE INTEL SCC

The first experiments as presented in the next subsection concerned the impact of the underlying hardware. The second subsection presents and discusses scalability results obtained with the globally greedy vertex moving algorithm on a small and a large graph. All measurements are based on the clock configuration with 800MHz cores, 1600MHz on-chip network, and 1066MHz DDR memory.

A. Shared Memory Performance

Table I summarizes measurements of the memory access latencies following the WGR cost model [13]. The *write overhead* w is the time to issue a write, the *write gap* g is the time until the next write can be issued and the *read overhead* r is the time it takes to fetch data from the memory. For NCM the latencies where measured with scalar values (byte, short, int), while for MPBT, MPBT+WT, and DCM whole cache lines were used. For DCM write performance measurement the lines were read before writing to them because the SCC's caches have no allocate-on-write. Access to the off-chip memory takes around two times longer than to the on-chip memory.

To congest a memory controller, an increasing number of cores read concurrently from the controller. Figure 4 shows the impact on the read latency and congestion is visible above three concurrent readers. Up to three cores do not interfere with each other and, above that, the read overhead increases linearly. Thus, when spreading the data evenly over all four memory controllers, at least 12 cores are necessary to utilize the memory bandwidth. In comparison, the on-chip SRAM can handle more than 15 cores before congestion is visible [13].

The SCC has no hardware mechanism to flush the L2 cache. Thus, a system call has to be used to evict lines by reading other data. The costs of flushing unmodified and modified data are shown in Figure 5. Up to 128 lines the costs increase linearly with 8 000 cycles for one line and $300N + 7\,500$ for N lines. Flushing a 4kB page (128 lines) takes around 47 100 cycles and the entire cache needs 582 000 cycles. Writing back a modified line requires around 80 additional cycles.

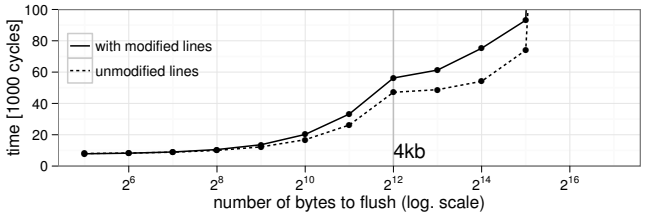


Figure 5. Costs of manually flushing unmodified lines from the L2 cache.

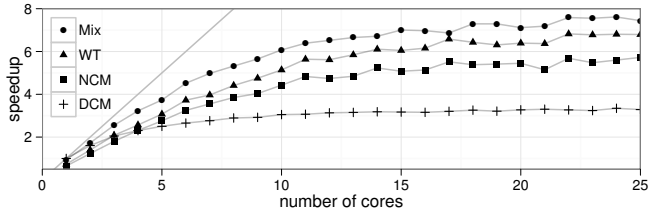


Figure 6. Speedup with the WorldImport1999 graph relative to the single-core DCM-BC variant.

For comparison, a TACO collective operation over all cores of the SCC usually completes within 8 000 cycles [13]. Thus, this cache flushing always takes much longer than the invalidation broadcasts and, therefore, the differences between the three consistency controllers of Section III-B are negligible.

B. Globally Greedy Vertex Moving

For a varying number of cores, the algorithm was applied on a relatively small graph that represents world trade relations in the year 1999. The graph has just 66 vertices and 4290 edges [20]. The local search took 64 steps and we measured the overall time of these steps without the program initialization and reading the data file. The measurements considered the three map implementations DCM, NCM, and WT from Section III-C and a *Mix* variant, which uses DCM static data (e.g. vertices, edges, weights) and WT for maps that are modified by vertex moves (e.g. partitions, partition weights).

To compare the measurements, Figure 6 shows the speedup factors relative to the DCM variant on a single core. Only the results with the Broadcast consistency controller are shown, because the other two controllers were similar. Using only DCM maps, the local search scales really poorly. This is probably caused by the huge cache flushing overhead at the master and the worker cores. With NCM maps better speedups are achieved because almost all of the cache flushing is eliminated. However, this variant does not use any caching and indeed seems to utilize the memory bandwidth when reaching 12 cores. The WT maps improve on this because they use the L1 caches and achieves speedup 6 with 13–14 cores. The *Mix* variant produced the best results with speedup 7 at 15 cores. More importantly, for small core counts the speedup was consistently better than with the other variants.

Amdahl's law provides an upper bound to the achievable speedup. It is based on the parallel runtime $T(P) = T_s + T_p/P$ and the speedup $T(1)/T(P)$, where T_s is the sequential work,

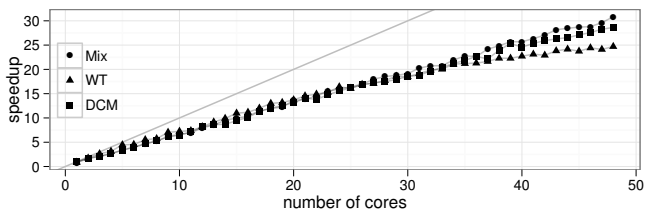


Figure 7. Speedup with the Patents graph. All speedup factors are relative to single-core DCM-BC.

e.g. applying the best vertex move, and T_p/P is the parallel work with P cores, e.g. evaluating the modularity gains of all vertices. However, overheads impact the speedup, leading to

$$T(P) = T_s + T_p/P + T_{wb} + T_{inv} + \alpha P + \beta \log_2(P) \ .$$

After each vertex move, the master worker has to write back all modifications to the main memory, which costs T_{wb} in total. Then, all workers invalidate these memory ranges in their caches with overhead T_{inv}. Because this flushing is done in parallel, it can be attributed once to the sequential costs. The workers might perform some work that is effectively sequentialized by memory bottlenecks. This can be modeled by αP because this sequential work increases with the number of workers. Finally, the workers are invoked in each step by using a multicast tree, which introduces a logarithmically growing coordination overhead $\beta \log_2(P)$.

Fitting linear models on the measurements indicated that the coordination overhead $\beta \log_2(P)$ is negligible, while αP is necessary to explain the low slope at low core counts. The NCM, WT and Mix variants eliminate $T_{wb} + T_{inv}$, and WT and Mix reduce αP through caching. To improve the speedup bound, it is necessary to reduce αP by increasing the memory bandwidth or reducing the cache misses. The latter requires caching of more data by using DCM maps, which is only efficient if the flushing overheads $T_{wb} + T_{inv}$ can be reduced.

With 66 vertices, the world trade graph is quite small, e.g. each worker is responsible for just 5 vertices when using 12 workers. Thus, we repeated some of the measurements with the much larger NBER U.S. patent citations graph, which has 240 547 vertices [21]. Figure 7 compares the runtime of the first 500 steps. Here, the task size was large enough to dominate even the flushing overheads of the DCM variant.

V. RELATED WORK

Two other shared memory mechanisms are available on the SCC. POPSHM provides a simple put/get copy-based interface to access shared memory. It does not remap unused LUT entries but uses a few as read/write buffer in NCM mode. The SMC library supports allocation of shared pages, changing the access modes, and provides release consistency with configurable consistency domains. Both libraries do not use remote method invocation mechanisms, which limits their flexibility. In contrast, we implemented the shared memory management and consistency control together with the application on top of a common messaging subsystem.

Implementing scaleable cache coherence protocols is still challenging because directories grow with the cache size and the number of cores [5]. Some try to balance parameters to reduce the directory size while also keeping the coherence traffic low [22]. In comparison, we omitted the directories and used broadcasts at very coarse granularity by exploiting the algorithm's structure. Other projects exploit typical sharing patterns to compress the directories [23], [24]. Software-level coherence is an interesting alternative because it can incorporate knowledge about the application at design time and can have better performance than hardware coherence in some cases [7]. A promising mixture of both approaches is the Cohesion memory model of Kelm et al. [6].

VI. CONCLUSIONS

We presented a framework for software-level cache coherence on the SCC. A simple parallel graph partitioning algorithm was used to evaluate the impact of the software-level cache coherence and the speedups achievable through parallelization. The experiments showed that considerable speedups are possible depending on the problem size despite non-negligible cache flushing overheads. The results could be improved by better hardware support for manual cache control. The presented framework would benefit from a write-back and a write-back-invalidate instruction on logical address ranges. On architectures with a shared cache level, the write-backs and invalidations would be necessary only on the private levels. Porting the data structures from an existing application was mostly straightforward, but the composed structures had to be changed considerably to interact efficiently with the consistency framework. From a software engineering perspective this might actually not be a drawback, because moving operations on shared data into the access proxies also decouples independent types of operations. For example, the graph initialization methods were implemented in a separate access proxy.

ACKNOWLEDGMENTS

We thank Intel for the access to the SCC and MARC program. In particular, we thank Michiel W. van Tol (University of Amsterdam), Werner Haas (Intel Research Braunschweig), and Jan-Arne Sobania (HPI Potsdam) for implementing the software-based L2 cache flushing. This research would not have been possible otherwise.

REFERENCES

[1] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, pp. 67–77, May 2011.

[2] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and Improvements of Programming Models for the Intel SCC Many-core Processor," in *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS2011), Workshop on New Algorithms and Programming Models for the Manycore Era (APMM)*, Istanbul, Turkey, July 2011.

[3] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, pp. 18:1–18:15, 2008.

[4] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, pp. 15–31, 2007.

[5] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, "Directory-based cache coherence in large-scale multiprocessors," *Computer*, vol. 23, no. 6, pp. 49–58, Jun. 1990.

[6] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: a hybrid memory model for accelerators," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA '10, 2010, pp. 429–440.

[7] S. V. Adve, V. S. Adve, M. D. Hill, and M. K. Vernon, "Comparison of hardware and software cache coherence schemes," in *Proceedings of the 18th annual international symposium on Computer architecture*, ser. ISCA '91, 1991, pp. 298–308.

[8] 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.

[9] X. Zhou, H. Chen, S. Luo, Y. Gao, S. Yan, W. Liu, B. Lewis, and B. Saha, "A Case for Software Managed Coherence in Many-core Processors," Poster on 2nd USENIX Workshop on Hot Topics in Parallelism HotPar10, 2010.

[10] Y. P. Zhang, T. Jeong, F. Chen, H. Wu, R. Nitzsche, and G. R. Gao, "A study of the on-chip interconnection network for the ibm cyclops64 multi-core architecture," in *Proceedings of the 20th international conference on Parallel and distributed processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 64–64.

[11] A. Olofsson, "A 1024-core 70 GFLOP/W Floating Point Manycore Microprocessor," Poster on 15th Workshop on High Performance Embedded Computing HPEC2011, 2011.

[12] 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.

[13] 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.

[14] 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.

[15] R. Rotta and A. Noack, "Multilevel local search algorithms for modularity clustering," *J. Exp. Algorithmics*, vol. 16, pp. 2.3:2.1–2.3:2.27, Jul. 2011.

[16] M. E. J. Newman, "Analysis of weighted networks," *Physical Review E*, vol. 70, p. 056131, 2004.

[17] ——, "Finding community structure in networks using the eigenvectors of matrices," *Physical Review E*, vol. 74, p. 036104, 2006.

[18] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner, "On modularity clustering," *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, no. 2, pp. 172–188, 2008.

[19] S. L. Min and J.-L. Baer, "Design and analysis of a scalable cache coherence scheme based on clocks and timestamps," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 1, pp. 25–44, Jan. 1992.

[20] A. Noack, "Example graphs from the LinLogLayout tool," http://www-sst.informatik.tu-cottbus.de/~an/GD/, 2008.

[21] V. Batagelj and A. Mrvar, "Pajek datasets," http://vlado.fmf.uni-lj.si/pub/networks/data/patents/Patents.htm, 2006.

[22] A. Gupta, W.-D. Weber, and T. C. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *Proceedings of the 1990 International Conference on Parallel Processing (ICCP)*, vol. 1: Architectur, 1990, pp. 312–321.

[23] H. Zhao, A. Shriraman, and S. Dwarkadas, "Space: sharing pattern-based directory coherence for multicore scalability," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10, 2010, pp. 135–146.

[24] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009, pp. 423–434.

 ISBN: 978-2-7257-0016-8

One-Sided Communication in RCKMPI for the Single-Chip Cloud Computer

Steffen Christgau and Bettina Schnor
Institute of Computer Science, University of Potsdam
August-Bebel-Strasse 89, 14482 Potsdam, Germany
Email: {christgau, schnor}@cs.uni-potsdam.de

***Abstract*—One-Sided Communication functions have been defined in the Message Passing Interface standard since version 2. Modern implementations of the interface, such as MPICH2, support One-Sided Communication. This paper presents insights to the MPICH2 architecture and the implementation of One-Sided Communication in its low-level CH3 communication modules. Further, issues for using MPI's One-Sided Communication features on the Single-Chip Cloud Computer are presented and resolved. The paper also presents a comparative scalability study of an example application both on the SCC and on an InfiniBand cluster.**

I. INTRODUCTION

The Message Passing Interface (MPI) [1] is a commonly used programming API to implement parallel applications. Version 2 of the MPI Standard [2] specifies synchronous, asynchronous, one-sided, and two-sided communication, where one-sided operations are defined as being asynchronous by default.

MPI two-sided communication calls are due to the classic send-recv scheme where each send call must have a matching receive call at the destination. In contrast, One-Sided Communication (OSC) defines a process interaction mechanism where only *one* process specifies the communication parameters. The other process (called *target*) only has to provide the memory area (called *window*) and is not required to call any communication routine.

MPI-2-OSC separates communication from synchronization and defines so called synchronization *epochs*. An arbitrary number of communication calls can be synchronized within a single epoch. This bundling reduces the need to make synchronization calls for each transfer. MPI-2 provides several API calls to open and close such an epoch. Figure 1 shows some pseudo code of the typical use of one sided and two sided communication.

The remainder of the paper is organized as follows: First, an overview of the architecture of MPICH2 and the implementation of MPI's OSC method in the low-level CH3 devices is presented. Based on this discussion, issues in the MPI implementation for the SCC are revealed and fixed. The corrected implementation is then used in Section V to compare the scaling of an example CFD application both on the SCC using one- and two-sided communication as well as on an InfiniBand cluster. The conclusion and an overview of related work constitute the end of the paper.

```
 Node A                          Node B
-------------------------------------------------
One-sided example:

 MPI_Win_Create                  MPI_Win_Create

 MPI_Win_Post                    MPI_Win_Start
  <computation>                   <computation>
                                  MPI_Put
                                  <computation>

 MPI_Win_Wait                    MPI_Win_Complete
-------------------------------------------------
Two-sided example:

                                 <computation>
  MPI_Irecv                      MPI_Isend
  <computation>                  <computation>
  MPI_Wait                       MPI_Wait
```

Fig. 1. Pseudo code examples of unidirectional asynchronous data exchange via one-sided and two-sided communication.

II. MPICH2 ARCHITECTURE

MPICH2 is a portable MPI implementation [3]. Its modular architecture is shown in Figure 2. The methods defined by the MPI standard are implemented on top of third generation of the abstract device interface (ADI3) [4]. This layer wraps around the message transportation device of the target platform and thereby provides hardware-independence. A downside of this interface is the huge amount of functions that have to be implemented for new platforms. To facilitate the adaption on new hardware platforms the CH3 device was introduced. This device implements the ADI3 and presents a much simpler interface to so-called channels. The channels implement the CH3 interface and are responsible to receive and send messages from resp. to the hardware.

Within an CH3 channel implementation several requirements and conventions from the higher layers have to be considered. First, all MPI processes are considered to be connected by virtual connections which do not have to manifest in physical connections. Device specific information for a connection between two processes, a socket e.g., can be attached to the according virtual connection. If a message is going the be sent to a remote process, the upper layers pass a request to the CH3 channel which activates the according virtual connection. If it is not possible to satisfy the request

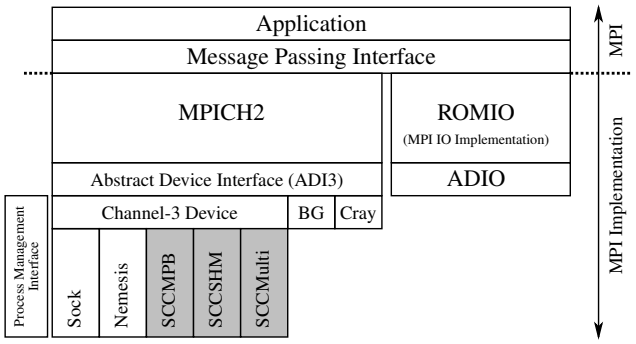


Fig. 2. MPICH2's layered architecture

immediately, that is the message was not transferred at all or only a partial transfer was possible, the channel is responsible to try a retransmission of the unsent part at a later time. This can be done by adding the request to a queue associated to the virtual connection.

To send outstanding messages, the CH3 device invokes a so-called progress engine which must be implemented by the channel. The implementation checks whether requests have been enqueued and if so, it tries to send these messages or even fragments. With the invocation of the progress engine, the channel must also check for new messages that have been received by the hardware.

If the channel has detected a new message, it asks upper MPICH2 layers to create a new receive request based on the received data. As soon as more data for this request is received by the channel, it must check whether the request has been completed by the incoming data. To do so, call-back functions are invoked. Besides a default completion call-back function, a request may provide its specific routine that has to be called by the channel. This notification scheme also applies to the requests created when sending messages. Due to the callback mechanism, the decision whether the request has been completed successfully is not up to the channel itself, but to the higher MPICH2 layers. The channel implementation must therefore be aware of changes in the send resp. receive requests.

III. OSC WITH CH3 DEVICES

MPICH2 does not require a CH3 channel to implement any specific function to support one-sided communication. Instead, the only requirements for a fully-functional channel is to implement some book-keeping functions, four methods to start sending a new message resp. processing a send request and one function that represents the progress engine. As the latter one is responsible for receiving new messages, there are no explicit receive methods.

With these minimal requirements, a CH3 channel is able to process send and receive requests issued by point-to-point operations like MPI_SEND and MPI_RECV as well as collective routines such as MPI_BCAST or MPI_GATHER. Moreover, the functions to be implemented by a channel also enable the usage of one-sided communication. This is made possible by the CH3 device which provides an implementation for the OSC-related functions of MPI resp. the ADI3 layer. This implementation is based on point-to-point operations. Thus, a channel device is not required to implement OSC-functions, since the CH3 device breaks the according calls down to an appropriate call sequence of point-to-point primitives.

The implementation of the OSC functions in the CH3 device basically relies on a queue. The operations issued by communication calls like MPI_PUT, MPI_GET and MPI_ACCUMULATE and initial locking methods like MPI_WIN_LOCK are stored in the queue. By calling a final synchronization method, like MPI_WIN_UNLOCK, the enqueued operations are sent to the target process by the same mechanism used for point-to-point communication.

The synchronization calls use this mechanism as well. For locking a window with MPI_WIN_LOCK, for example, a request to lock the window is sent to the target process. After that, the progress engine is invoked until a message is received which grants access to the lock.

To address hardware that offers support for OSC, e.g. by remote direct memory access (RDMA), the CH3 device allows channels to overwrite the default OSC implementation. This flexibility is achieved through function pointers within the internal structure that represents a remote memory access (RMA) window. During the window creation, any of these pointers can be overwritten by the CH3 channel implementation. Thereby, MPICH2 can be instructed to use the channel's implementation of OSC synchronization or communication calls.

IV. OSC SUPPORT ON THE SCC

MPICH2 had been ported to the Single-Chip Cloud Computer (SCC) by implementing three different CH3 channels. The port is named RCKMPI and is based on MPICH2 1.2.1p [5]. All of the SCC-specific CH3 channels use shared memory which is either backed by the message passing buffer (MPB), the external memory or both. For the latter case, the memory used for the messages is switched at a constant threshold. Although the memory used for message transport differs, the implementation is basically similar: The memory is partitioned into sections such as each process resp. core offers dedicated write areas for every other MPI process. Regarding bandwidth, the channel that uses the MPB delivers best performance.

As all channels implement the functions required by the CH3 device, all communication methods are expected to work. On the one hand, this is actually true for point-to-point operations. On the other hand, one-sided communication is not usable with the original RCKMPI implementation. For instance, the SCC-specific CH3 channels only cover some basic cases of the MPICH2 test suite. That way, RCKMPI applications are likely to crash if this communication paradigm is used.

There are several reasons for this issue. First, the RCKMPI source code always calls the default message completion handler defined by MPICH2. Thus, RCKMPI does not account the case where a special completion call-back function has been associated with the send request. As those call-backs are only defined when OSC-related messages should

be transferred, applications based on RCKMPI will show unpredictable behaviour when using OSC.

Even if the appropriate call-back is invoked on message completion, the channel implementation is not aware of request modifications caused by upper MPICH2 layers. These modifications are mainly applied when OSC messages are transferred. As a consequence, even when invoking the completion method, OSC applications are again likely to crash.

Therefore, the following implementation issues were fixed for the MPB CH3 channel implementation of RCKMPI: (a) Invoke the message completion call-back function when send and receive requests are assumed to be completed. (b) Check that the call-back handler confirms the completion of the request. (c) If the call-back signals an incomplete request, further process the unfinished request. (d) On message reception, copy its payload back to the according user buffers. The fixed source code is available to the MARC community [6].

V. EXPERIMENTAL RESULTS

Based on this work, a CFD-like cellular automaton [7] was used to compare the applications scalability on the SCC with an InfiniBand cluster. The cellular automaton is a 9-point-stencil computation. We tested two different versions of the application: one uses OSC and the other two-sided routines from the MPI standard. The programs use the communication patterns shown in Figure 1. The communication is performed after each time step: every MPI process has to exchange 8 KB of data with its two neighbor processes. The according messages are sent before the process starts to calculate the inner values of its field fraction to gain optimal computation-communication-overlap.

In the experiment, the application was executed with an increasing number of nodes while the size of the computed two-dimensional field was fixed. Since there are only 28 nodes in the InfiniBand cluster, the scaling was only analyzed up to a number of 28 nodes for both systems. Each node of both the SCC and the InfiniBand cluster executed a single MPI process. The runtime of the sequential version is divided by the runtime of parallel program running on n processors to obtain the speedup.

A. Experimental Environment

The InfiniBand cluster consists of 28 machines equipped with two Intel Xeon 5520 CPUs each providing four cores (Hyperthreading disabled). On each node, 48 GB of DDR3 main memory are installed. The InfiniBand connection is based on 20 Gb/s Mellanox MHGH19-XTC ConnectXZ cards and a Mellanox MTS3600R-1UNC switch. Further, OpenMPI 1.4.3 was used as an MPI-2 implementation that supports InfiniBand. During the experiments, OpenMPI's parameters were choosen such that OSC functions use InfiniBand's RDMA features. It is therefore not expected that the OSC application on the SCC scales better than on the InfiniBand cluster as there is no RDMA support in RCKMPI.

On the SCC, a modified Linux 2.6.38.3 kernel was used in conjunction with Busybox. For compiling the application and RCKMPI, GCC version 4.5.2 has been chosen. On the InfiniBand cluster, Scientific Linux with Kernel 2.6.18 and GCC 4.3.2 is installed.

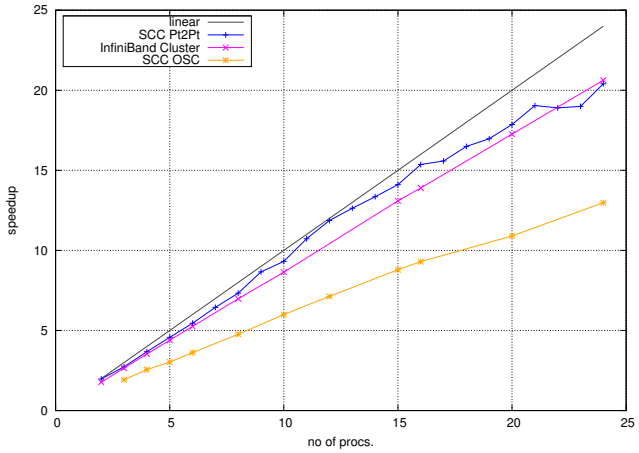


Fig. 3. Speedup of the CFD application compared to its sequential version on the SCC and the InfiniBand Cluster

B. Results on SCC

The obtained results are presented in Figure 3. The OSC version on the SCC shows linear scaling, but it is outperformed by the two-sided communication version on the SCC. The speedup values are about one third smaller than the two other versions and only one half of the optimal linear speedup.

The scaling of both the two-sided and the OSC version of the application can be attributed to the low latency communication network and the RCKMPI CH3 channel implementation. A downside of this implementation is the inherited handling of OSC by the CH3 device implementation. That is queueing all communication actions until a final synchronization call is issued (see Section III).

By processing the queue entries messages are sent for locking the window, transferring the data and finally unlocking the window. Thus, at least three MPI message are sent to the target process in a synchronous manner. Compared to the single non-blocking communication call (MPI_Isend) that is used in the point-to-point version of the application two additional messages are required. On the SCC, the two additional messages are sent with an handshake protocol between the communication processes, whereas on the InfiniBand cluster RDMA does not require any participation of the target process.

C. Comparison with InfiniBand Cluster

The OSC version running on the InfiniBand cluster reached approximately linear speedup (see Figure 3). This shows the benefit of an RDMA capable network which allows the implementation of communication libraries which support computation-communication-overlap.

VI. RELATED WORK

The authors of [8] report up to 30 % runtime improvement for an atmospheric simulation, when using MPI-2 one-sided

communication in combination with OpenMP instead of point-to-point communication. It is not clear if the improvement is due to the use of OpenMP or one-sided communication or the combination.

Different implementation options for MPI2-OSC were analysed in the literature regarding the different capabilities of the underlying networks [9]–[14]. While one would expect inefficiencies of OSC over transports without support for remote direct memory access (RDMA), interconnects like InfiniBand are expected to offer similar or better performance for OSC than for two-sided communication. However, asynchronous two-sided communication still offers the best performance if only a few number of communications are synchronised with one epoch [9], [10], [15].

In [14] the authors observe that one-sided communication performs much worse than two-sided communication for short and medium-sized messages. The reason is spotted in the overhead of synchronization functions.

The same observation is also made in [15], [16]. A comparison of two- and one-sided MPI2 communication over Gigabit Ethernet [15] shows that the design of the MPI2-OSC API is the key performance problem. The design of MPI2-OSC was compared with another OSC API called NEON in [16] on top of InfiniBand networks. While InfiniBand is an ideal network for computation-communication-overlap, the MPI2-OSC implementation again suffered from the additional overhead and was outperformed by NEON. The authors show that applications which use MPI-2-OSC suffer from the overhead of the additional synchronisation message that has to be sent in order to complete a remote memory access.

Currently, the MPI community discusses the MPI-3 standard [17]. The One-Sided communication interface proposed by the MPI-3 RMA Working group retains all of the calls from MPI-2, but adds new additional calls for window creation, synchronisation, and communication. Some of the proposed new features are implemented and investigated in [18]. Especially, the new Request-Based operations seem to be promising to achieve optimal computation-communication-overlap.

VII. CONCLUSION AND FUTURE WORK

In this paper, we gave insights on the implementation of One-Sided Communication in MPICH2 which is the base for the MPI implementation of the SCC – RCKMPI. Based on this insights, we identified missing functionalities within that implementation. These issues were fixed and the revised source code was made available to MARC members.

Further, the scalability of a CFD-like application based on the OSC implementation of RCKMPI was analyzed and compared to both a two-sided version on the SCC and a one-sided version on an InfiniBand cluster. It was revealed that all variants scale in a linear fashion, while the two-sided version on the SCC outperforms the two others. Moreover, the OSC variant shows poor scaling behavior on the SCC. On the other hand, the good scaling of the OSC version on the InfiniBand cluster is obviously due to the opportunity to use the RDMA features. This effect has already been observed when comparing the MPICH2 implementation with the lightweight OSC API called NEON both on Gigabit Ethernet [15] and InfiniBand installations [16].

Future work goes into two directions: First, the scaling of the OSC features of RCKMPI can be improved. As the SCC offers hardware support for defining and accessing shared memory areas, a MPI window for OSC might be defined with help of the hardware features: A window may reside in a shared memory which is backed either by the MPB or the external DDR3 main memory. Further, accesses of a MPI process resp. core could be supported by dedicated cores. For example one core per tile could be responsible for performing RMA operations while the other handles computational tasks of an application [19]. That way, both RMA and overlap of communication and computation would be made possible.

Second, it would be interesting to get experiences with applications using the Global Arrays toolkit on the SCC. This framework offers a "virtual shared memory programming interface." Moreover, the implementation is based on message passing libraries, such as MPICH2 and requires OSC features. Especially, application from the field of quantum chemistry are heavily using frameworks like Global Arrays [20].

REFERENCES

[1] Message Passing Forum, "MPI: A Message-Passing Interface Standard," Tech. Rep., 1994.

[2] ——, "MPI-2: Extensions to the Message Passing Interface," Tech. Rep., Jul. 1997.

[3] W. Gropp, "MPICH2: A new start for MPI implementations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, D. Kranzlmueller, J. Volkert, P. Kacsuk, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2002, vol. 2474, pp. 37–42.

[4] W. Gropp and E. Lusk, "MPICH abstract device interface version 3.3," Mathematics and Computer Science Division Argonne National Laboratory, Reference Manual Version 3.3, Dec. 2001, draft.

[5] I. Comprés Ureña, M. Riepen, and M. Konow, "RCKMPI – lightweight MPI implementation for Intel's Single-chip Cloud Computer (SCC)," in *18th European MPI Users' Group Meeting, EuroPVM/MPI 2011*, ser. Lecture Notes in Computer Science, Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2011, vol. 6960, pp. 208–217.

[6] S. Christgau, "MARC Bugzilla Bugreport 386: Missing support for one-sided communication," http://marcbug.scc-dc.com/bugzilla3/show_bug.cgi?id=386, Feb 2012.

[7] "The Cellular Automaton Application," http://www.cs.uni-potsdam.de/bs/cellularautomat/, 2006.

[8] A. A. Mirin and W. B. Sawyer, "A scalable implementation of a finite-volume dynamical core in the community atmosphere model," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 3, pp. 203–212, 2005.

[9] B. Barrett, G. M. Shipman, and A. Lumsdaine, "Analysis of Implementation Options for MPI-2 One-Sided," in *Proceedings of the 14th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science, vol. 4757. Springer-Verlag, September/October 2007, pp. 242–250.

[10] W. Gropp and R. Thakur, "An evaluation of implementation options for mpi one-sided communication," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface 12th European PVM/MPI User's Group Meeting*, D. M. et al., Ed. Sorrento, Italy: Springer-Verlag, September 2005, pp. 415–424.

[11] W. Huang, G. Santhanaraman, H.-W. Jin, and D. K. Panda, "Design alternatives and performance trade-offs for implementing mpi-2 over infiniband," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface 12th European PVM/MPI User's Group Meeting*, D. M. et al., Ed. Sorrento, Italy: Springer-Verlag, September 2005, pp. 191–199.

[12] T. S. Woodall, G. M. Shipman, G. Bosilca, R. L. Graham, and A. B. Maccabe, "High Performance RDMA Protocols in HPC," in *Proceedings of the 13th European PVM/MPI Users' Group Meeting*, ser. Lecture

Notes in Computer Science, vol. 4192. Berlin, Heidelberg: Springer-Verlag, September 2006, pp. 76–85.

[13] J. Liu, W. Jiang, H. Jin, D. Panda, W. Gropp, and R. Thakur, "High performance mpi-2 one-sided communication over infiniband," in *International Symposium on Cluster Computing and the Grid (CCGrid 04)*, April 2004.

[14] Rajeev Thakur and William Gropp and Brian Toonen, "Optimizing the Synchronization Operations in MPI One-Sided Communication," *High Performance Computing Applications*, vol. 19, no. 2, pp. 119–128, 2005.

[15] L. Schneidenbach and B. Schnor, "Design issues in the implementation of MPI2 one sided communication in Ethernet based networks," in *PDCN'07: Proceedings of the 25th conference on Parallel and Distributed Computing and Networks (PDCN)*. ACTA Press, Feb. 2007, pp. 277–284.

[16] L. Schneidenbach, D. Böhme, and B. Schnor, "Performance Issues of Synchronisation in the MPI-2 One-Sided Communiction API," in *15th European PVM/MPI Users' Group Meeting*. Dublin, Ireland: Spinger, Lecture Notes in Computer Science 5205, 2008, pp. 177 – 184.

[17] MPI 3.0 Standardization, "MPI: A Message-Passing Interface Standard, Draft," http://meetings.mpi-forum.org/MPI_3.0_main_page.php, Nov. 2010.

[18] S. Potluri, S. Sur, D. Bureddy, and D. K. Panda, "Design and implementation of key proposed mpi-3 one-sided communication semantics on infiniband," in *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, ser. EuroMPI'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 321–324. [Online]. Available: http://dl.acm.org/citation.cfm?id=2042476.2042514

[19] 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.

[20] J. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju, "Supporting the Global Arrays PGAS Model Using MPI One-Sided Communication," in *IPDPS 2012*, Shanghai, China, May 2012, in press.

 ISBN: 978-2-7257-0016-8

Asynchronous Broadcast on the Intel SCC using Interrupts

Darko Petrović, Omid Shahmirzadi, Thomas Ropars, André Schiper
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
firstname.lastname@epfl.ch

***Abstract*—This paper focuses on the design of an asynchronous broadcast primitive on the Intel SCC. Our solution is based on OC-Bcast, a state-of-the-art k-ary tree synchronous broadcast algorithm that leverages the parallelism provided by on-chip Remote Memory Accesses to Message Passing Buffers. In the paper, we study the use of parallel inter-core interrupts as a means to implement an efficient asynchronous group communication primitive, and present the userspace library we designed to be able to use interrupts in OC-Bcast and make it work asynchronously. Our experimental evaluation shows that our algorithm allows parallel broadcast operations to efficiently progress concurrently and provides low latency for a single broadcast operation. It highlights that parallel interrupts can help implementing efficient group communication primitives on many-core systems.**

I. INTRODUCTION

Recent research in microprocessor design indicates that the most promising way to achieve high performance while lowering power consumption is to integrate many loosely-coupled processors on a single chip [1]. A many-core chip can be viewed as a distributed system, i.e. a set of cores connected through a Network on Chip (NoC). The Intel Single-Chip Cloud Computer (SCC) is a 48-core research prototype of a many-core chip, designed to be operated as a message passing system.

Group communications, such as broadcast, are of major importance in message passing systems, and have been widely studied in different contexts. Considering the low latency and high throughput of a NoC, a many-core chip is very similar to a parallel High Performance Computing (HPC) system. However, results show that porting an HPC communication library to the SCC requires rethinking the design of the communication algorithms [10].

In this paper, we study the implementation of an asynchronous broadcast primitive for the Intel SCC. Our previous work, done in the context of Single Program Multiple Data (SPMD) applications, studied synchronous broadcast operations [10]. It shows that leveraging specific features of the Intel SCC, *i.e.*, Remote Memory Access (RMA) to on-chip Message Passing Buffers (MPB), helps improving the performance of group communications by increasing parallelism in the data dissemination. We adapt the resulting algorithm, called OC-Bcast (On-Chip Broadcast), to work asynchronously in order to be able to use it in a more general execution model. To do so, we propose to use parallel Inter-Processor Interrupts (IPI).

The paper presents the following contributions:

- A study of the global interrupt controller (GIC) on the Intel SCC, and the description of a library to simply manipulate IPIs in userspace (Section IV).
- An asynchronous version of OC-Bcast based on parallel IPIs that allows arbitrary interleaving of concurrent broadcast operations (Section V).
- An evaluation of the proposed algorithm showing that it manages to achieve both low single broadcast latency and high concurrent broadcasts throughput, demonstrating usefulness of parallel IPIs in implementing efficient group communication on many-core chips (Section VI).

Before detailing the contributions, we describe the SCC in Section II and focus on the related work on interrupt-based communication and broadcast on the SCC in Section III.

II. THE INTEL SCC

The SCC is a general purpose many-core prototype developed by Intel Labs. In this section we briefly describe the SCC architecture and inter-core communication.

a) Architecture: The cores and the NoC of the SCC are depicted in Figure 1. There are 48 Pentium P54C cores, grouped into 24 tiles (2 cores per tile) and connected through a 2D mesh NoC. Tiles are numbered from (0,0) to (5,3). Each tile is connected to a router. The NoC uses high-throughput, low-latency links and deterministic virtual cut-through X-Y routing [5]. Memory components are divided into (i) message passing buffers (MPB), (ii) L1 and L2 caches, as well as (iii) off-chip private memories. Each tile has a small (16KB) on-chip MPB equally divided between the two cores. The MPBs allow on-chip inter-core communication using Remote Memory Access (RMA): Each core is able to read and write in the MPB of all other cores. There is no hardware cache coherence for the L1 and L2 caches. By default, each core has access to a private off-chip memory through one of the four memory controllers, denoted by MC in Figure 1. In addition, an external programmable off-chip component (FPGA) is provided to add new hardware features to the prototype.

b) Inter-core communication: To leverage on-chip RMA, cores can transfer data using the one-sided *put* and *get* primitives provided by the RCCE library [8]. Using *put*, a core (a) reads a certain amount of data from its own MPB or its private off-chip memory and (b) writes it to some MPB. Using *get*, a core (a) reads a certain amount of data from some MPB and (b) writes it to its own MPB or its private off-chip memory. The unit of data transmission is the cache line, equal

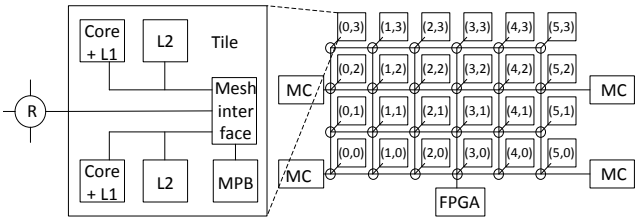


Fig. 1: SCC Architecture

to 32 bytes. If the data is larger than one cache line, it is sequentially transferred in cache-line-sized packets. During a remote read/write operation, each packet traverses the routers on the way from the source to the destination.

Cores are also able to notify each other using inter-process interrupts (IPI), either by writing directly into the receiving core configuration register or by using the Global Interrupt Controller (GIC)[1]. In the latter case, the receiving core is able to obtain additional information about the interrupt through a set of GIC registers. We consider the GIC in this work.

III. Related Work

In the SCC context, there are works on interrupt-based message passing ([6], [7], [9], [11], [12]), as well as on the implementation of collective operations ([4], [2], [8]). However, to the best of our knowledge, there is no work combining the two, that is, leveraging IPIs for collective communication. For this reason, we present the related work in two categories: (i) papers that focus on the collectives, i.e. broadcast and (ii) those that discuss interrupt-based communication.

A. Broadcast Algorithms

Despite several implementations of broadcast on the Intel SCC, the only scenario considered so far is running HPC applications. This assumes the SPMD model, in which each core runs the same program and every core explicitly invokes a routine to participate in a collective operation. As a consequence of this assumption, polling can be used for notification and asynchronous primitives are not necessary.

When it comes to the broadcast algorithms used, RCCE_comm [3], as well as RCKMPI [2] use well-known algorithms based on two-sided communication – binomial tree and scatter-allgather. On the other hand, OC-Bcast [10] applies a tree based algorithm for broadcast directly on top of *put*/*get* primitives, which dramatically improves both latency and throughput by minimizing memory copy operations on the critical path. The algorithm presented in this paper has been directly derived from OC-Bcast, as described in Section IV.

B. Communication Based on Interrupts

The assumption of having only one program running at a time, as well as synchronous communication among cores, which holds for HPC applications, is not valid in general-purpose distributed systems. Therefore, using interrupts for asynchronous communication is a must for porting such systems to the SCC.[2] Examples of SCC software relying upon inter-core interrupts are numerous ([6], [7], [9], [11], [12]). In the context of this paper, most interesting works are those that give specific details on different ways of using interrupts and their cost in terms of performance.

The SCC port of Barrelfish [9] uses IPIs to notify cores about message arrivals. The round-trip message latency reported by the authors was found too high for point-to-point communication in such a system, despite running it on bare metal with the minimum needed software overhead.

Another approach for leveraging interrupts, using the GIC, has been applied in the SCC port of distributed S-NET [12], a declarative coordination language for many-core chips. The port is based on an asynchronous message-passing library: Interrupts are trapped by the Linux kernel and then forwarded to the registered userspace process in the form of a UNIX signal, which is the idea reused in this paper. Using a similar round-trip experiment as in [9], the authors confirm the high latency of inter-processor interrupts. Moreover, the latency they observe is even higher than in [9], mainly because of a necessary context switch before delivering a signal to the registered userspace process. A direct comparison with RCCE, the native SCC message passing library based on polling [8], has shown that IPIs are far less efficient in terms of latency for point to point communication.

Despite being costly for point-to-point message passing, IPIs can be used for asynchronous collective communication with an acceptable cost, as we show in this work.

IV. Broadcast Based on Interrupts

This section describes the design and implementation of our broadcast library based on inter-processor interrupts (IPI). First, we give an overview of the underlying hardware mechanism for sending parallel interrupts. Then we briefly describe OC-Bcast, a polling-based broadcast algorithm for the SCC, and explain how we have adapted it to use interrupts instead.

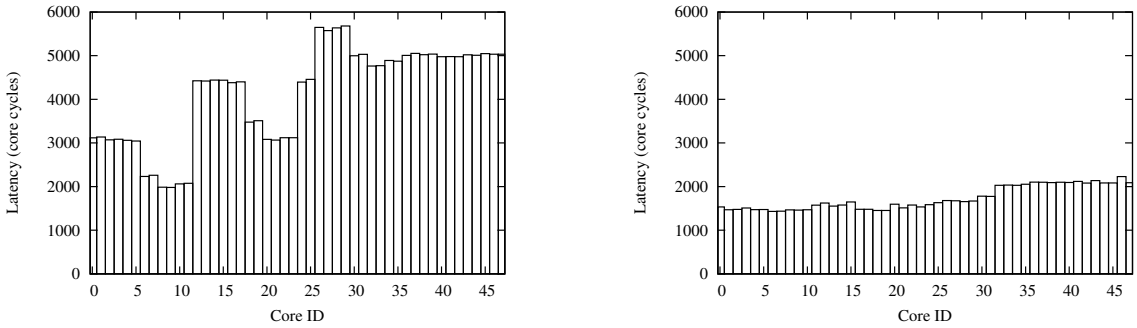
A. Interrupt Hardware on the SCC

Using the basic IPI mechanism on the SCC, a core can send an interrupt to another core by writing a special value to the configuration register of that core. This generates a packet which is sent through the on-chip network to the destination core. Although this mechanism is simple and straightforward, it lacks some essential features. For example, the identity of the notifier is unknown and it is possible to send only one interrupt at a time.

Fortunately, the SCC has an off-chip FPGA, which allows for adding new hardware features. An extension to the basic IPI mechanism has been provided by Intel, which comprises a set of registers for managing IPI (request, status, reset and mask). As a consequence, a core can send an interrupt to up to 32 other cores in just one instruction, by writing an

[1]The GIC is available starting with sccKit 1.4.0 and is located on the FPGA.

[2]Strictly speaking, it is possible to communicate asynchronously using a dedicated polling thread, but this solution wastes CPU cycles and energy.



(a) With access to GIC registers

(b) Without access to GIC registers

Fig. 2: Latency of broadcasting an interrupt at the kernel level

appropriate bit mask to its request register[3]. The work of generating interrupt packets is completely delegated to the FPGA interrupt controller.

To test whether the FPGA interrupt controller actually delivers multiple interrupts in parallel, we have performed the following experiment: A core sends an interrupt to all cores (including itself), by issuing two instructions which write a mask of "1"-s to its request register on the FPGA. Then, the core measures the time until it receives its own interrupt. The results, given in Figure 2a, indicate a significant difference in latency observed by different cores, ranging from about 2000 to almost 6000 core cycles (cf. VI-A for setup details). Further experiments have confirmed that this difference grows as a function of the number of cores that the interrupt is sent to – it is barely noticeable for less than 20 cores, but then starts to increase rapidly.

The experiment presented above could lead us to the conclusion that parallel notification using interrupts scales poorly, but further investigation explains this result. Namely, upon receiving an interrupt, there is a fixed set of steps a core should perform. This includes reading from the status register, to determine the sender, and resetting the interrupt by writing to the reset register. Since all the registers related to interrupt handling are on the FPGA, access to them is handled sequentially. When an interrupt is sent to many cores at once, they all try to access their interrupt status register at the same time, but their requests contend and are handled one after another, which explains the observed performance loss. We believe that a proper on-chip implementation of interrupt registers would eliminate this problem, since they could be accessed in parallel. To confirm that the reason for bad scaling of the interrupt mechanism is contention on the FPGA, we have repeated the same experiment, but this time deliberately avoiding the FPGA registers, except on the sending core. In Figure 2b we see that the times measured across the cores are very similar and close to 2000 core cycles. Slight differences in latency are easy to explain. Namely, the FPGA is connected to the mesh via the router between tiles (2,0) and (3,0) (cf. Figure 1), so the round-trip time to the FPGA is shorter for cores closer to this router. Next, it takes slightly more time for cores 32 to 47 to receive their interrupt. This is because, as already described, it is possible to send at most 32 interrupts by issuing a single instruction. Therefore, when broadcasting an interrupt, a core first broadcasts to cores 0 to 31 in the first instruction, and then to the other cores, which results in slightly higher latency.

Another set of experiments, as well as comparisons with results of other authors [12], confirmed that the latencies presented in Figure 2b are practically indistinguishable from the latency of sending point-to-point interrupts (about 2000 cycles). This implies that the cost of notification using interrupts is practically constant with respect to the number of cores notified. However, as we have described, sequential access to the off-chip registers for interrupt handling slows down the whole process in the current implementation on the SCC. Still, from Figure 2a we can see that even with this effect, broadcasting an interrupt to the 48 cores is only about 3 times more expensive than sending a point-to-point interrupt, making this mechanism interesting for use in group communication.

B. OC-Bcast Based on Interrupts

Now we describe how the SCC interrupt hardware presented above can be used to perform asynchronous broadcast. As the base, we used OC-bcast [10], an optimized on-chip broadcast algorithm built on top of one-sided put and get primitives. The principle of OC-bcast is the following: a broadcast k-ary tree is formed, with the sender as its root. The sender puts the message in its MPB and notifies its k children, which then copy the message to their own MPBs in parallel and notify the parent that it can free its MPB. The children repeat this process with their children, until all the cores have got the message. The value of k is configurable. Obviously, higher values of k offer more parallelism, but they can lead to contention on the MPB, which can cancel out the gain obtained by the increase in parallelism. This is not a problem for the SCC itself (OC-Bcast with $k = 47$ even gives the lowest latency for some message sizes), but can be an issue at large scale.

However, in its original flavor, OC-Bcast uses MPB polling for notification. Each child has a flag in its MPB that it polls when waiting for a message. This means that the children

[3]The upper limit of 32 is merely a consequence of the 32-bit memory word on the P54C

cannot be notified in parallel about the existence of a message, since the parent can write only one flag at a time, which was mitigated to some extent by using a special notification tree. This problem can be addressed by parallel interrupts. The modified algorithm can be summarized as follows:

1) The sender puts the message from its private memory to its MPB and sends a parallel interrupt to all its children. Then it waits until all the children have received the message.
2) Upon receiving the interrupt, a core copies the data from the parent's MPB to its own MPB and acknowledges the reception of the message to the parent by setting the corresponding flag in the parent's MPB.
3) The core then sends a parallel interrupt to notify its own children (if any) and then copies the message from the MPB to its private memory. Then it waits until all its children have received the message.
4) When all core's children have acknowledged the reception, the core can make its MPB available for other actions (possibly a new message).

C. Implementation

To implement the modified OC-Bcast, we have developed a userspace library for interrupt handling, following the idea given in [12]. Namely, a userspace process can register itself with a special kernel module. Every time an interrupt from another core is received, the kernel module sends a real-time UNIX signal to the registered process, which triggers a user-provided handler. We have opted for real-time signals because they can be queued if there is more than one signal pending. This way, we ensure that every interrupt is converted to a signal and the algorithm can be written entirely in userspace.

A drawback of this approach is a performance loss already observed in [12], since it increases the end-to-end delay of sending interrupts. Namely, the numbers presented in Figure 2b show only the latency until the receiver's kernel handles the interrupt. To propagate it to a userspace process in the form of a UNIX signal, a context switch is necessary, which significantly increases the cost. Nevertheless, we have adopted this approach for two reasons. Firstly, such an implementation changes only absolute numbers and does not prevent us from observing changes in performance resulting from design-level decisions. The same algorithm could be implemented in the Linux kernel or directly on bare metal, which completely avoids UNIX signals and context switching. Secondly, our library is easy to integrate with RCCE and the accompanying tools, which makes it convenient for other researchers willing to use inter-processor interrupts without significant effort.

V. MANAGING CONCURRENT BROADCAST

OC-Bcast was initially designed in the context of SPMD applications, where a core has to explicitly call the broadcast function to participate in the collective operation. As a consequence, a core is involved in only one collective operation at a time. Using interrupts in OC-Bcast allows us to move to a more general model where broadcast operations can arbitrarily interleave at one core. In this section, we study how to efficiently manage this aspect.

The algorithm described in Section IV-B has to be modified to allow asynchronous broadcast operations issued by different cores. Indeed, without modifications the algorithm would be prone to deadlocks. A simple scenario can be used to illustrate a deadlock situation. Consider two cores c and c' that try to broadcast a message concurrently, with c' being a child of c in the tree where c is the root and the opposite in the tree where c' is the root. Core c' cannot copy the message that c is trying to broadcast in its MPB because it is busy with its own message. Core c' will be able to free its MPB when it knows that all its children have copied the message. However c cannot get the message from c' either, because it is in exactly the same situation as c'. There is a deadlock.

To deal with this problem, a simple solution would be to use a global shared lock to prevent multiple broadcast operations from being executed concurrently. In this case, the problem becomes equivalent to broadcast in the SPMD model and no further modifications to OC-Bcast are necessary. However, this would limit the level of parallelism and prevent us from fully using the chip resources.

To avoid deadlocks without limiting the parallelism, we adopt the following solution: If the MPB of some core c is occupied when a notification about a new message arrives, c copies the message directly to its off-chip private memory. Additionally, if c has to forward the message, it is added to a queue of messages that c has to forward. Eventually, when the MPB is available again, c removes messages from the queue and forwards them to the children.

Algorithm 1 presents the pseudo-code of this solution for a core c. In the presented algorithm, we do not put any requirements on the tree structure. We only assume that a predefined deterministic algorithm is used to compute the broadcast trees. Thus, during the initialization, each core is able to compute the tree that will be used by each source (line 7). Furthermore, if a message is larger than the available MPB, it is divided into multiple chunks.

For the sake of simplicity, the pseudo-code is not fully detailed. It only illustrates the important modifications that are made to avoid deadlocks. We define three functions as an interface to the algorithm described in Section IV: $OCBcast\_send\_chunk(chunk, Tree)$ initiates the sending of the chunk $chunk$ in the tree $Tree$; $OCBcast\_receive\_chunk(chunk, buf, src)$ allows to get $chunk$ from the MPB of core src in buf, buf being either the MPB of the caller or a memory region in its off-chip private memory; $OCBcast\_forward\_chunk(chunk, Tree)$ is used to forward a $chunk$ in the tree $Tree$. Contrary to $OCBcast\_send\_chunk()$, $OCBcast\_forward\_chunk()$ assumes that the chunk is already in the MPB of the sender. In the pseudo-code, a chunk includes not only payload, but also some meta-data, *i.e.*, the id of the core that broadcasts the message ($chunk.root$) and the id of the message the chunk is part of ($chunk.msgID$).

As mentioned before, we allow a core to receive chunks directly in its off-chip private memory when its MPB is busy with another chunk that is being sent (line 17). Thus, the

Algorithm 1 Asynchronous broadcast algorithm (code for core c)

Local Variables:
1: MPB_c {MPB of core c}
2: $MPBStatus_c \leftarrow$ `available` {Status of the MPB}
3: $chunkQueue_c \leftarrow \emptyset$ {Queue of chunks to forward}
4: set of trees $Tree_1, Tree_2, ..., Tree_n$ {$Tree_c$ is the tree with c as root}

5: **initialization:**
6: define **deliver_chunk()** as the IPI handler
7: **for** $coreID \in 0...n$ **do** compute $Tree_{coreID}$

8: **broadcast**(msg)
9: **for all** chunk of msg **do**
10: **broadcast_chunk**($chunk$)

11: **broadcast_chunk**($chunk$)
12: $MPBStatus_c \leftarrow$ `busy`
13: **OCBcast_send_chunk**($chunk$, $Tree_c$)
14: $MPBStatus_c \leftarrow$ `available`
15: **flush_queue()**

16: **deliver_chunk**($chunk$, $source$)
17: **if** $chunkQueue_c$ is empty $\wedge$ $MPBStatus_c =$ `available` **then**
18: $MPBStatus_c \leftarrow$ `busy`
19: **OCBcast_receive_chunk**($chunk$, MPB_c, $source$)
20: **if** c has children in $Tree_{chunk.root}$ **then**
21: **OCBcast_forward_chunk**($chunk$, $Tree_{chunk.root}$)
22: $MPBStatus_c \leftarrow$ `available`
23: **flush_queue()**
24: **else**
25: let $item$ be the memory allocated to receive the chunk
26: **OCBcast_receive_chunk**($chunk$, $item$, $source$)
27: **if** c has children in $Tree_{chunk.root}$ **then**
28: enqueue $item$ in $chunkQueue_c$
29: **if** msg corresponding to $chunk.msgID$ is complete **then**
30: deliver msg to the application

31: **flush_queue()**
32: **while** $chunkQueue_c$ is not empty **do**
33: dequeue $chunk$ from $chunkQueue_c$
34: $MPBStatus_c \leftarrow$ `busy`
35: **OCBcast_send_chunk**($chunk$, $Tree_{chunk.root}$)
36: $MPBStatus_c \leftarrow$ `available`

sender can free its MPB. The chunks that the core is supposed to forward to other cores, are stored in a queue (lines 25-28), that is flushed when the MPB becomes available (line 15 and line 23). Note that to ensure fairness, if the MPB is free at the time the core receives an interrupt but some chunks are already queued to be forwarded (line 17), the chunk is received in the private memory and added to the queue. Thus, a chunk cannot overtake another chunk that has been in the queue already for some time. However, if no chunk is in the queue and the MPB is available, the chunk is first copied in the MPB to limit the number of data movements between the MPB and the private memory that could harm the performance of the broadcast operation [10].

VI. EVALUATION

In this section we evaluate our broadcast algorithm. After describing the system parameters used for our experiments, we measure the latency of the presented broadcast algorithm and compare it with that of OC-Bcast. Then we show how the algorithm behaves with different values of k and with more cores broadcasting at the same time.

Message Size (Number of cache lines)	1	32	64	128
OC-Bcast	44.0 μs	76.1 μs	112.6 μs	189.8 μs
Asynchronous broadcast	40.2 μs	75.5 μs	118 μs	196.7 μs

TABLE I: Comparing the latency of synchronous broadcast (OC-Bcast) and asynchronous broadcast for different message sizes.

A. Setup

We have performed the experiments under the default SCC settings: 533 MHz tile frequency, 800 MHz mesh and DRAM frequency and standard LUT entries. We use sccKit 1.4.1.3, running a custom version of sccLinux, based on Linux 2.6.32.24-generic. The kernel of every core runs the special kernel module for converting interrupts to UNIX signals, described in Section IV.

B. Experiments

The first experiment measures the latency when messages of different sizes are broadcast from one core (core 0 in this case). We fix the value of k to 47 (see Section IV), which enables us to obtain the highest level of parallelism when sending the interrupts and reading from the MPB. Due to space constraints, we do not consider other values of k in this experiment.

Table I compares the obtained latency with that of OC-Bcast[4]. The two algorithms have very similar latencies with these settings. This confirms that the interrupt hardware on the SCC is useful for designing asynchronous collective operations, even though its latency is high for point-to-point communication, as pointed out in other studies [9], [12].

It is interesting to notice that the latency of the asynchronous broadcast algorithm increases faster as a function of the message size. This is because of a higher level of MPB contention. More specifically, it is pointed out in [10] that too much parallelism in accessing the MPB can impair performance. In OC-Bcast, notifications are propagated using a binary tree, which results in less overlapping accesses to the MPB of the sender than when a parallel interrupt is sent. This shows that extremely high values of k might be inappropriate at large scale because of the contention effect.

In the second experiment, we change the output degree of the broadcast tree (k) and the number of sources, that is, the number of cores broadcasting in parallel. Each source repeatedly broadcasts a 4 KB (128 cache lines) message from its private memory, without waiting for the other cores to receive the message, thus creating a message pipeline. This way we observe the throughput of the system, that is, the amount of data broadcast in a unit of time.

The result of this experiment is given in Figure 3. With a single source, the throughput decreases as k increases. The reason is the cost of polling flags (there are at most k flags to poll). To wait for an acknowledgment from its children, each parent has to poll k flags in its MPB and reset them afterwards. The variations in the performance can be explained by the fact

[4]The version of OC-Bcast considered here is slightly optimized with respect to the original paper which presents it [10].

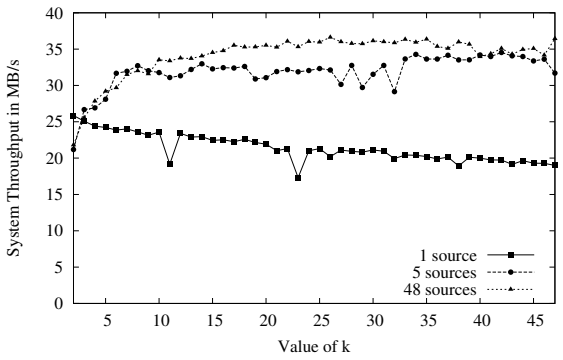


Fig. 3: Throughput of the asynchronous broadcast algorithm for different values of k and different number of concurrent sources

that a core does not control when it will be signaled. In fact, when a core is about to forward a received message to the children, it can get interrupted to receive another message. If this happens, the children have to wait, which introduces sporadic performance drops.

With more than one source, the throughput increases. There are two possible reasons for this. The first one is that when a single node is broadcasting messages, the other cores are sometimes idle waiting for the next message to be available. With multiple sources, this idle time can be used to receive messages from other sources. The second reason is that if a core receives interrupts in different trees, it can often have more than one interrupt waiting to be serviced by the kernel. When this happens, all the pending interrupts will be serviced (converted to signals) one after another, and only then will the execution switch back to the userspace process. This actually means that there will not be one context switch per interrupt, but significantly less, resulting in performance increase.

We can also see that the difference in throughput when broadcasting from 5 and 48 sources is not significant. This is because the system gets saturated. Based on the model presented in [10], the maximum bandwidth when copying data from a core's MPB to the off-chip memory is about 55 MB/s (assuming cache line prefetching implemented in software as in iRCCE [4]). Our algorithm achieves 68% of this maximum bandwidth.

When it comes to the choice of k with multiple sources, the trend is opposite to the single-source case. This is especially visible for smaller values of k, where each increase by 1 evidently increases the throughput. To understand this, recall that the resources of every core are effectively used in this case, in the sense that there is no idle time. However, performing a broadcast operation consumes more resources on different cores if k is lower since there are more interrupts to send. Thus, the cores manage to do less useful work.

C. Discussion

The presented experiments show two important properties of our asynchronous broadcast algorithm. First, in spite of being built on more general assumptions, its latency is comparable with that of the most efficient synchronous broadcast algorithm currently available for the SCC. Second, the algorithm manages concurrent broadcasts efficiently, even when all cores are broadcasting at the same time.

VII. CONCLUSION

In this work we have presented a novel asynchronous broadcast algorithm for the Intel SCC, which is based on RMA and parallel IPI. Our algorithm is derived from OC-Bcast, an optimized synchronous broadcast algorithm for the SCC. The evaluation of our asynchronous broadcast primitive demonstrates that the algorithm manages to efficiently deal with concurrent broadcast operations to achieve low latency and high system throughput. Comparisons with existing synchronous broadcast primitives also show that parallel IPI are of general interest to implement efficient group communications on many-core chips.

As future work, we plan to study the use of IPI and on-chip RMA operations for other group communication primitives on the Intel SCC. Especially, we will focus on group communication primitives that provide ordering properties, to implement replicated data structures.

REFERENCES

[1] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[2] I. C. Urena, M. Riepen, and M. Konow. RCKMPI–Lightweight MPI Implementation for Intel's Single-chip Cloud Computer (SCC). *Recent Advances in the Message Passing Interface*, pages 208–217, 2011.

[3] E. Chan. RCCE_comm: a collective communication library for the Intel Single-chip Cloud Computer. http://communities.intel.com/docs/DOC-5663, 2010.

[4] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and improvements of programming models for the Intel SCC many-core processor. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 525 –532, july 2011.

[5] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks (1976)*, 3(4):267–286, 1979.

[6] S. Lankes, P. Reble, C. Clauss, and O. Sinnen. The Path to MetalSVM: Shared Virtual Memory for the SCC. In *The 4th symposium of the Many-core Applications Research Community (MARC)*, page 7, 2011.

[7] N. Linnenbank, F. Reader, A.S. Tanenbaum, and D. Vogt. Implementing MINIX on the Single Chip Cloud Computer. www.nieklinnenbank.nl/download/scc.pdf, 2011.

[8] T. Mattson and R. Van Der Wijngaart. RCCE: a Small Library for Many-Core Communication. *Intel Corporation, May*, 2010.

[9] S. Peter, A. Schpbach, D. Menzi, and T. Roscoe. Early experience with the Barrelfish OS and the Single-Chip Cloud Computer. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany*, 2011.

[10] Darko Petrović, Omid Shahmirzadi, Thomas Ropars, and André Schiper. High-Performance RMA-Based Broadcast on the Intel SCC. In *24th ACM Symposium on Parallelism in Algorithms and Architectures*, Pittsburg, PA, USA, June 2012. to appear.

[11] R.F. van der Wijngaart, T.G. Mattson, and W. Haas. Light-weight communications on Intel's single-chip cloud computer processor. *ACM SIGOPS Operating Systems Review*, 45(1):73–83, 2011.

[12] M. Verstraaten, C. Grelck, M.W. van Tol, R. Bakker, and C.R. Jesshope. Mapping distributed S-Net on to the 48-core Intel SCC processor. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany*, 2011.

 ISBN: 978-2-7257-0016-8

Work in Progress: Malleable Software Pipelines for Efficient Many-core System Utilization

Janmartin Jahn, Sebastian Kobbe, Santiago Pagani, Jian-Jia Chen, Jörg Henkel
Karlsruhe Institute of Technology (KIT), Germany

***Abstract* — This paper details our current research project on the efficient utilization of many-core systems by utilizing applications based on a novel kind of software pipelines. These pipelines form malleable applications that can change their degree of parallelism at runtime. This allows not only for a well-balanced load, but also for an efficient distribution of the cores to the individual, competing applications to maximize the overall system performance. We are convinced that malleable software pipelines will significantly outperform existing mapping and scheduling solutions.**

***Index Terms* — Parallel architectures, multicore processing, pipeline processing, multiprocessing systems.**

I. INTRODUCTION AND MOTIVATION

Running multiple applications efficiently on a many-core system requires careful decisions about the distribution of the availablecores among the running applications becauseof the following reasons: an inefficient distribution of coresmay lead to an imbalanced load, thus leaving resources idle, or to a reduced system performance (even though the load may be perfectly balanced) when assigning few cores to demanding applications while giving more coresto applications that hardly benefit from them. Thelatter may be due to different application *efficiencies*(i.e. the speed-up per core), which areexpressed bythe corresponding *speed-up functions* that describe how the performance of an application dependson the number of cores assigned to it.Thus, thesedecisions largely impact the overall performanceof many-core systems. Parallel applications often are not able to achieve a linear speed-up with the number of cores [1], i.e. the efficiency decreases. Consequently, when running multiple applications, it is crucialto distribute the cores in a way that the overall efficiency is maximized, which is illustrated in Figure 1 for the combined (system) efficiency when running two competing applications.

There are three kinds of parallel applications: a) statically parallelizedapplications that are only able to execute on a fixed number of cores, b) moldable applications whose degree of parallelism can be defined at the start-up time of the application,and c) malleable applications that can change their degree of parallelism (from now on, we will call this *resizing*) at runtime.With varying and/or dynamic workloads, malleable applications allow the highest overall system efficiency as the optimal number of cores could be assigned to each application whenever the system state changes [2].

However, the effort and the costs of resizing have to be considered because they may be manifold and include task migration and workload distribution, which may be especially hurtful for many-core architectures with distributed memories [3]. Depending on the kind of application, resizing is not possible at arbitrary points. Today, malleable applications are often used in high-performance computing and grid-computing environments, often work on large datasets, and run for long periods of time (e.g., hours or days). Here, the absolute time required for the re-distribution of cores is of minor importance as the total system throughput in long periods of time is crucial. However, due to this property, such applicationsare not suitable for highly interactive systems or mobile devices: here, systems need to be responsive, so the time spent for resource re-distribution needs to be minimal and theefficiency of the system needs to be improved quickly (and not in the long term) to satisfy user demands. In typical high-performance computing environments, large re-distribution timesare tolerable whencompared to the total application runtime. Instead, we address systems where applications must beresized at a low overhead to allow frequent variations – e.g., when new applications enter the system,the system state changes, or a user starts interacting with the device.

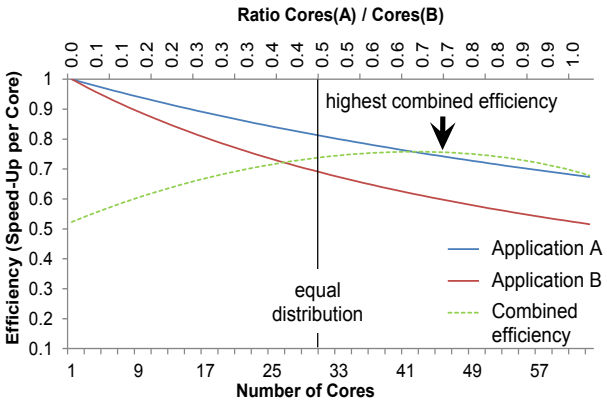


Figure 1 - Combined efficiencies of two competing applications

To make decisions about which application should be allowed to use how many resources, state-of-the-artresource management schemes for malleable applications (such as [4, 5]) need to know how well the application will perform with these resources, i.e. thespeed-up function must be known. This knowledge could be obtained by offline profiling or online

monitoring. After distributing the coresamong the runningapplications, the application itself has to balance the loadamong its assigned cores.

Our novel contributions are malleable software pipelines that addressthe three issues of frequent malleability, unknown speed-up functions, and autonomous load balancing.Thisincludes: a) software pipelines that are malleable through decreasing and increasing their level of parallelism at runtime by *fusing*stages, i.e. combining multiple consecutive stages into one larger stage, or splitting previously fused stages (we call this *fission*), b) an online monitoring framework that decides upon the best option for *fusion* and *fission* operations, and c) a compiler that creates those software pipelines using an architecture specification and application profile with a maximum level of granularity.

II. RELATED WORK

There are multiple ways to create malleable applications, but basically they are based on the same principles. On shared memory architectures, creating malleable applications can be done by (e.g., compiler based [6] or library based [7]) exploitation of thread and even loop-level parallelism. The costs for resizing the application are comparably low (because no data has to be migrated between private memories) and allow frequent resizing of the application at the granularity of several hundred milliseconds. For distributed memory systems, where each core has its own, private memory, these approaches are not suitable because the overhead for resizing applications can be prohibitively high. Thus, approaches like Master-Slave parallelization are used [8]. Here, depending on the amount of available cores, more or less workers are created, i.e. resizing is possible at thread level, but the resizing costs are much higher than in a shared memory system.

Consequently, the time between resizing decisions has to be long enough so the gain from the increased efficiency can exceed the corresponding overhead, which makes them very suitable for large scientific computing environments where applications run for hours or days, but less suitable for interactive systems running on MPSoCs. Another possibility is a single program multiple data (SPMD) application architecture [9], where the data is partitioned depending on the available cores. Adjusting the data partition to changing system states is possible but comes at a high cost, which also only allows for seldom changes in the application size.

Adaptive-MPI (AMPI) [10] provides a multitude of 'virtual cores'as the smallest level of parallel granularity, which are prepared at compile time and are mapped to physical coresat runtime. Typically, multiple virtual cores are mapped on one real core. AMPI is based on the Charm++ language [11] and utilizes the same runtime system for load balancing as applications written in Charm++. Resizing of applications implemented with AMPI is transparent to the application itself as only the mapping of virtual to real cores has to be changed and, if necessary, the working set has to be transferred. Therefore, AMPI cannot take advantage of application-specific knowledge (such as choosing optimal migration points between iterations of pipeline stages). AMPI is designed for large, distributed computing environments (e.g., datacenters) where the individual nodes have large main memories and fast cores. In contrast to this, we target architectures with comparably slow individual cores where the access to (off-chip) main memory is very costly and must be reduced to a minimum.

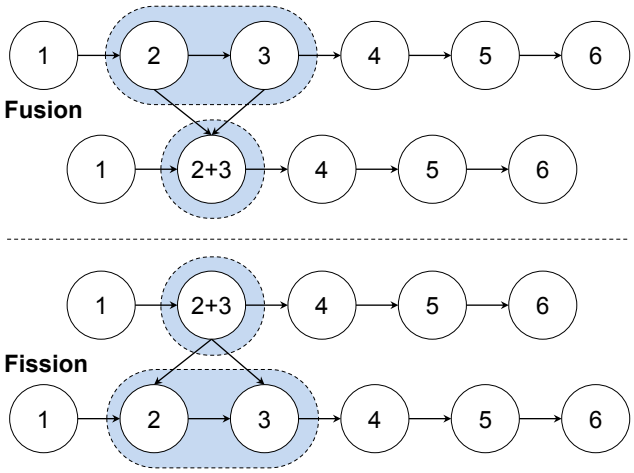


Figure 2 – Basic operations of malleable pipelines

III. OUR APPROACH

Software pipelines present a widely used programming model which is especially suitable to parallelize sequential complex applications for many-core systems with distributed memories that may be private to each core. Multiple compilers, tools and frameworks exist that extract software pipelines from existing applications [12-14]. The presented malleable software pipelines are software pipelines with the following properties:

They support the basic operations of *fusion* and *fission*, as illustrated in Figure 2. A fusion of pipeline stages reduces the degree of parallelism, thus reducing the number of stages by combining two consecutive stages into one. Contrarily, fission increases the degree of parallelism by splitting fused stages.Pipelines are created with a compile-time chosen finest level of granularity, from which no further fissions are possible. Stages can be fused until only one stage remains. In this case, the pipeline is equivalent to the sequential execution of the same algorithm.Malleable software pipelines use runtime monitoring to decide which stages to fuse or which stage to split.

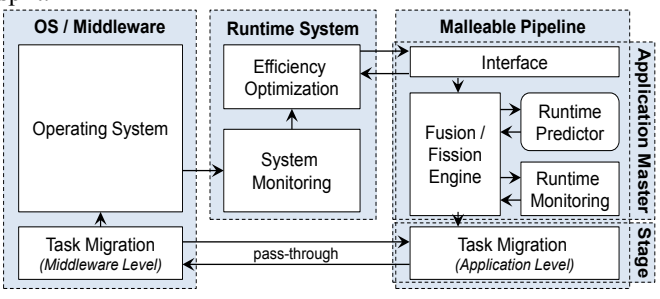


Figure 3 - Block diagram of system components

The system components are illustrated in Figure 3: The runtime system constantly monitors the system utilization and application efficiencies. An efficiency optimization component periodically re-evaluates the current system state and may decide to change the distribution of the system's cores to the applications. Each application consists of one so-called *Application Master* and one process per pipeline stage. These processes are based on the same executable file and may run one or more consecutive pipeline stages. When running multiple stages on one core, no inter-core data communication for these stages is necessary.

The *Application Master* includes an interface for the communication with the runtime system, a *Fusion / Fission Engine*, and a lightweight runtime predictor that predicts the potential performance increase of a fission operation and the potential performance decrease of a fusion operation, and an application level task migration component.

The *Interface* is responsible for the communication with the run-time system and provides it with information such as the maximum level of parallelism (static) and the gradient of the speed-up function (dynamic).

The *Fusion / Fission Engine*is responsible for conducting the fusions and fissions requested by the runtime system. The runtime system specifies which cores may be used or will be taken away from the application, while the Fusion / Fission Engine is responsible for identifying the stages to be fused or split.This decision is based on Runtime Monitoring of the computational and communication demands (from now on, we will call both computational and communication demands 'resource demands') of each individual stage. Using a lightweight Runtime Predictor, the Application Master decides which fusion will result in the least performance degradation and which fission will result in the highest performance increase. To avoid bloated communication volumes and races for communication hardware (such as the Message Passing Buffer in Intel's Single-Chip Cloud Computer [15]), only consecutive stages are fused. When the number of cores assigned to an application changes, all pipeline stages of that application are recombined.

The migration of the pipeline stages is very lightweight as they are migrated after completing an iteration, as software pipelines typically have a peak memory consumption while performing one iteration, and carry little state between iterations. As the executable file is the same for each stage and may dynamically switch from executing one stage to executing several stages, only the state information of the corresponding stages that is carried between iterations needs to be transferred. Consequently, the executable file needs to be transferred only when new cores are made available for an application and the stage fusions may be performed at runtime with very little overhead. In the presented case-study (see Section V), some stages were completely stateless (e.g., image enhancement or transformations which operate on each frame of the video stream individually) while others carried little state (e.g., information about identified objects in the previously processed image) in the worst-case of 38 kilobytes. The size of the state that needs to be transferred for each resizing decision depends on the executed application. However, we find that a typical property of software pipelines is that each stage carries comparably little state between iterations..

IV. RUNTIME SYSTEM

The runtime system is responsible for optimizing the efficiency of multiple applications running on a many-core system, thus competing for computational resources. This is performed by resizing the competing applications at runtime. To accomplish this, each application provides information about whether it could efficiently make use of more cores or if the already assigned cores are not used efficiently and a fusion of pipeline stages could be performed to free cores for other applications.If all possible applications and system states are known at design time, these decisions can be made at design time. This allows for optimal allocation, scheduling, and mapping of applications to cores. However, i.e. due to user interactions, changing input data, or unpredictable system state, an online resource management is required to efficiently utilize system resources.In [4], a distributed approach for managing malleable applications on many-core systems with hundreds or thousands of cores has been presented. The scalability of the approach is achieved by avoiding the use of global knowledge or broadcast communication. The approach utilizes an application performance model for parallel applications which considers the theoretical speed-up of an application [1] and the relative placement of the cores of the application to make its decisions.

In the following section we show how well the theoretical performance model and the performance of the real implementation of our malleable software pipelines match. By using real measurements of application performance and resizing overhead (which is currently not exactly modeled in the decision making process) we will be able to enhance the previous approach and apply it to a real many-core system running our malleable software pipelines.

V. CASE STUDY ANALYSIS

We analyze a software pipeline that we generated from a complex, real-world robotic application. The presented software pipeline captures stereo images from two pairs of stereo cameras, uses image enhancement and transformation algorithms, calculates a three-dimensional depth map, and then detects and tracks objects. Due to bandwidth constraints, the images can only be captured sequentially, not in parallel. The application tracks a recognized object and moves the robot to follow it. The 18 individual stages of the pipeline are illustrated in Figure 4. The case study focuses on the behavior of that software. To obtain the results, each pipeline stage had been benchmarked individually on the Intel SCC to obtain the necessary information to calculate the optimal fissions and fusions for any given number of cores.

When executing these stages on multiple cores, the speed-up function is largely sub-linear, as shown in Figure 5. A maxi-

mum speed-up of 9.7x can be achieved when running the application on 18 cores. The efficiency drops when increasing the number of cores, while positive "jumps" in efficiency when increasing the number of cores from 9 to 10 or from 11 to 12 are due to the fact that due to the increased number of cores, the fusions can be changed and otherwise heavily loaded cores have to carry much less load. The values represent stable states for different pipeline lengths. Dynamic effects (e.g., cache effects) that occur directly after resizing an application are not presented in this case study as the runtime system is still work-in-progress.

The overhead of stage fusions is limited as only the state that is carried between iterations needs to be transferred. For the applications we have examined, the magnitude of the required data size ranges from a few bytes to few kilobytes (with 38 kilobytes being the worst case), so the overhead of transferring this state is negligible in most state-of-the-art many-core architectures, which often have a high-bandwidth, low-latency inter-core communication fabric. The overhead of stage fission is purely dominated by transferring the executable file.

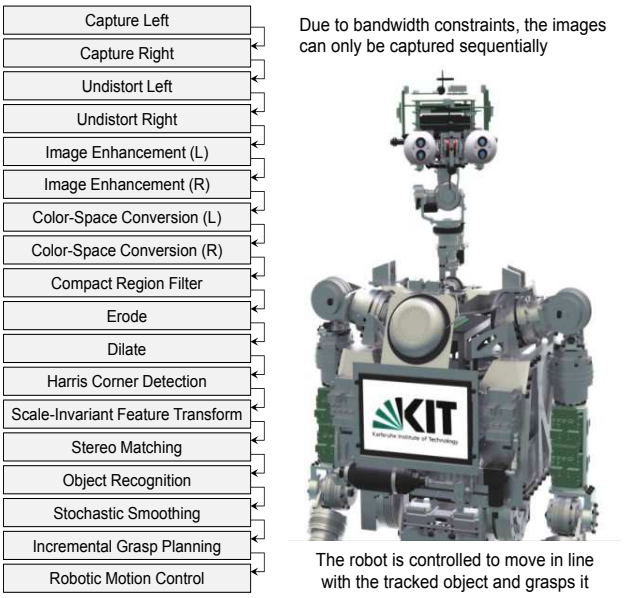


Figure 4 - Pipelined robotic application

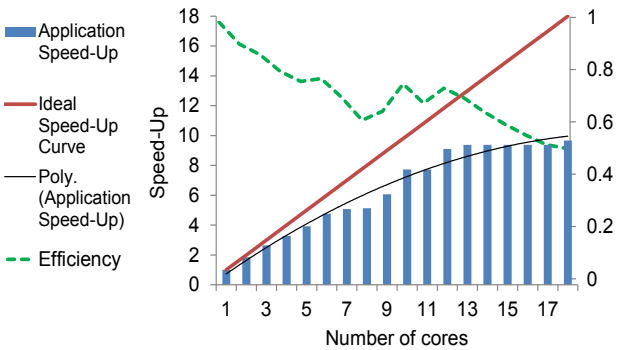


Figure 5 - Speed-up and efficiency of the robotic application

VI. OUTLOOK

The above argumentation urges that the efficiency of the utilization of the computational resources of many-core systems may be greatly enhanced when using malleable applications. In contrast to the state-of-the-art malleable application models, malleable software pipelines are well-suited to MPSoC architectures with comparably slow individual cores, small on-chip memories and highly penalized access to off-chip storage. Shifting malleability to the application layer allows software pipelines to change their degree of parallelism for efficiency optimization without incurring significant overhead because only the state that is carried across multiple iterations needs to be transferred.We are currently implementing the required infrastructure and conduct experiments that compare our malleable software pipelines to the state-of-the-art load balancing and task management systems.

REFERENCES

[1] A. B. Downey, "A parallel workload model and its implications for processor allocation," in *Sixth IEEE International Symposium on High Performance Distributed Computing*, August 1997, pp. 112–123.

[2] J. Hungershöfer, A. Streit, and J.-M.Wierum, "Efficient resource management for malleable applications",Technical Report, 2001.

[3] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual Design Automation Conference (DAC)*, 2007, pp. 746–749.

[4] S. Kobbe, L. Bauer, D. Lohman, W. Schröder-Preikschat, and J. Henkel, "DistRM: Distributed resource management for on-chip many-core systems," in *Proceedings of the IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct. 2011, pp. 119–128.

[5] P. Sanders and J. Speck, "Efficient parallel scheduling of malleable tasks," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, may 2011, pp. 1156 –1166.

[6] M. W. Hall and M. Martonosi, "Adaptive parallelism in compiler-parallelized code," *Concurrency: Practice and Experience*, vol. 10, no. 14, pp. 1235–1250, 1998.

[7] C. Pheatt, "Intel® threading building blocks," *J. Comput. Sci. Coll.*, vol. 23, no. 4, pp. 298–298, Apr. 2008.

[8] N. Islam, A. Prodromidis, and M. S. Squillante, "Dynamic partitioning in different distributed-memory environments," in *In Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 1996, pp. 244–270.

[9] L. V. Kalé, S. Kumar, and J. Desouza, "A malleable-job system for timeshared parallel machines," in *Proceedings of the 2nd International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, 2002, pp. 230–237.

[10] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *In Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, 2003, pp. 306–322.

[11] L. V. Kalé and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *In Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, 1993, pp. 91–108.

[12] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in C programs," in *International Symposium on Microarchitecture*, 2007.

[13] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, " Automatic Thread Extraction with Decoupled Software Pipelining," in *International Symposium on Microarchitecture*, 2005.

[14] J. Cheng, J. Castrillon, W. Sheng, H. Sharwachter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda, "MAPS: An Integrated Framework for MPSoC Application Parallelization," in *Design Automation Conference*, 2008.

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

 ISBN: 978-2-7257-0016-8

Enabling Computation Intensive Applications in Battery-Operated Cyber-Physical Systems

H.C. Woithe, W. Brozas, C.Wills,
B. Pichai, U. Kremer
Dept. of Computer Science
Rutgers University
Piscataway, NJ 08854
{hcwoithe, wbrozas, cwills,
bsp57, uli}@cs.rutgers.edu

M. Eichhorn
Institute for Automation
and System Engineering
Ilmenau University of Technology
98684 Ilmenau, Germany
mike.eichhorn@tu-ilmenau.de

M. Riepen
Intel Labs Braunschweig
Braunschweig, Germany
michael.riepen@intel.com

***Abstract*—Autonomous underwater vehicles (AUVs) have become indispensable tools for marine scientists to study the world's oceans. Real time examination of mission data can substantially enhance the overall effectiveness of AUVs in oceanography. However, current AUV technology only allows a detailed analysis of data after completion of a mission. The ability to perform onboard analysis of real time data is computationally intensive, requiring an energy efficient programming infrastructure that can be adapted to battery operated, energy constrained vehicles.**

Intel's 48-core SCC system exposes a collection of performance and energy/power knobs that can be refined for dynamically changing computation vs. energy tradeoffs. In this paper, we illustrate the potential benefits of these knobs for environment modeling and path planning. These applications are important for any autonomous cyber-physical system. Our experimental case study targets AUVs, particularly the Slocum glider. The results show that selecting different core, network, and memory controller speeds have a significant impact on the overall performance and energy requirements of our applications. Furthermore, the best selection is non-trivial and will depend on the available energy and computational needs of other mission critical tasks executing concurrently with modeling/path planning applications.

I. INTRODUCTION

Cyber-physical systems (CPSs) monitor events and conditions in the physical world, process information acquired through different sensors and input devices, and then determine a set of possible actions in response to the observed events and conditions. These systems often rely on battery power for long periods of time, so energy-aware data acquisition, data processing, and actuation is an important issue for increasing the lifetime and/or effectiveness of the overall system. There have been two major developments in recent years that have influenced the design and use of such autonomous systems; (1) the arrival of multi-core and soon many-core systems in the context of battery operated devices, and (2) the development and deployment of a variety of sensors and input/output (I/O) devices for such systems. Both trends are related since additional sensors or I/O require more computational power, and more computational power enables the use of additional, or more sophisticated sensors or I/O.

Multi-core systems have emerged that are designed to work in battery-operated devices. Such multi-core platforms provides substantial computational capabilities at low energy costs, making the execution of applications possible in autonomous environments that people did not believe possible only a few years ago. Recently, ARM announced the big.LITTLE system to improve energy efficiency of high-performance mobile platforms [1] while Intel has produced an experimental 48 core system called the SCC [2]. This system is not commercially available and has been designed to enable scalability research, particularly in the context of energy-aware computing. Allowing its cores, on-chip communication network and memory controllers to be dynamically configured with respect to supply voltage and/or frequency gives the SCC a wide range of performance and energy tradeoffs to support energy-efficient executions of mission critical applications.

In this work, we use ocean modeling (ROMS) and path planning applications for a buoyancy-driven autonomous underwater vehicle (AUV), the Slocum glider, to illustrate how the energy-aware features of the SCC could be used to react to changing energy vs. performance tradeoff requirements. Changes to these requirements can be triggered by the computational needs of other applications which are considered more mission critical resulting from the observation of an internal or external event. Clearly, avoiding obstacles has high priority when navigating through a busy shipping lane. Encountering a physical phenomenon like an algal bloom could also trigger the use of additional sensors and data processing applications. In addition, later phases of a long duration mission may have to deal with reduced battery power and energy budgets, putting more severe constraints on the applications that can be effectively executed. Our case study shows that

1) there are different performance/tradeoff points,
2) finding the best performance/tradeoff point during a mission is non-trivial, and also may change for different parts of a mission and their power caps, and
3) deploying a system such as the SCC in a battery-operated environment like an AUV can provide crucial computational capabilities.

Systems like the SCC could also be deployed in propeller-driven or hybrid AUVs in addition to buoyancy-driven gliders.

Fig. 1. One of our Slocum glider autonomous underwater vehicles equipped with a double payload bay and an acoustic modem. The shown configuration is 180 cm long and weighs about 90 kg.

Typically, propeller-driven systems have a significantly larger energy budget and a maximal mission duration of days to weeks, rather than weeks to months as the Slocum glider.

II. SLOCUM GLIDER

The Slocum glider belongs to a class of autonomous underwater vehicles that make use of a buoyancy engine, instead of a propeller, to traverse the ocean. The vehicle is a commercial product, manufactured by Teledyne Webb Research [3]. Buoyancy-driven flight, for the Slocum, is accomplished through the movement of a piston at the front of the vehicle. Retracting the piston causes the vehicle's displacement of water to decrease, thus allowing the glider to dive. Conversely, extending the piston increases displacement and enables the AUV to climb. Flight pitch can be fine-tuned through the movement of an internal battery pack. Along with wings and a controllable fin, the glider is able to navigate through the ocean at approximately 0.35 m/s [4]. A Slocum glider AUV on a benchtop is shown in Fig. 1.

The success of the glider as a research platform will depend on how well it can satisfy the increasing demands of ocean scientists for more and increasingly complex sets of sensors. Being capable of dynamically reacting to phenomena *in situ* is also becoming more common place. These requisitions necessitate increased computational capabilities. Current stock Slocum gliders are only equipped with two 16 MHz Persistor computers [5], one designated for the flight of the vehicle, while the other collects scientific data from sensors. In previous work, [6] several Linux single board computers (SBCs) were integrated and deployed within the AUV to track and dynamically adjust the vehicle's flight profile to fly within a thermocline off the coast of New Jersey. A system such as the SCC could be used in place of one of these SBCs to provide a flexible architecture that can effectively balance computation and energy requirements.

III. APPLICATIONS

Additional computational capabilities can save energy by more effectively managing the use of energy expensive sensors. The SCC is particularly well suited for this task because multiple energy saving algorithm/programs can run simultaneously on the chip, each with their own power and energy characteristics and tradeoffs. This section will provide an overview of such applications.

Dead Reckoning - Localization is a critical challenge for underwater operations. Typically, collected sensor data is tagged with spatial and temporal coordinates. AUVs can use GPS localization while at the surface, and dead reckoning (DR) while diving. Unfortunately, DR can result in significant localization errors in the presence of underwater currents. A Doppler Velocity Log (DVL) can be used to remedy this situation by performing bottom tracking, which allows the vehicle to measure its relative speed, thereby improving DR. However, operating the DVL sensor itself, and processing the acquired data can be energy and computation intensive. Without reliable localization many scientific missions are not feasible, including under-ice deployments where acquiring a GPS position at the surface is not possible.

Sensor Triggering - The Slocum glider does not currently support fine-grained or cross-sensor adaptive sampling. Sensors are typically turned on all the time, or active only on dives or climbs. The effectiveness of some sensors can be improved by making them part of a trigger chain, where low cost sensors activate more costly, but more precise sensors. Adaptive sampling may require significant physical modeling efforts and data processing capabilities.

ROMS - The Regional Ocean Modeling System (ROMS [7]) comprises a traditional ocean forecast model complemented by advanced variational data tools that allow the assimilation of 4-dimensional data, and more importantly, the sensitivity of the forecast to the present and future ocean state and the observational sampling pattern. For example, ROMS can be used to help optimize the path a glider takes between waypoints, or to indicate the regions where new observations would lead to the greatest improvement in forecast precision.

Charting the 3-dimensional and time varying pattern of these anomalies in ocean temperature and salinity represents an attractive test-bed for integrating ocean observation and simulation through adaptive sampling and smart control on a single platform. Optimizing the integrated system will necessitate trading off the sampling frequency, the sensors that are active, the distance traversed by the AUV, the ocean model computational effort, and communication, all of which make demands on the available battery power and energy.

Path Planning - The task of the path planning algorithm presented in this paper is to find a time-optimal path from a defined start position to a goal position while evading all static as well as dynamic obstacles in the area of operation, with consideration of the dynamic vehicle behavior and the time-varying ocean currents. The path planning algorithm, named the Time Variant Environment (TVE) algorithm [8], is based on a modified Dijkstra Algorithm [9]. A time-variant cost function is calculated during the search to determine the travel

times (cost values) for the examined edges of the graph. This modification allows the determination of a time-optimal path in a time-varying environment [8]. Proper path planning can be crucial if an AUV must arrive at a target location to observe a short lived phenomenon. It can also save time and energy since it allows the vehicle to navigate through strong ocean current fields.

IV. EVALUATION

As previously discussed, the SCC is particularly well suited for parallel applications. For this reason, the ROMS and path planning programs are targeted in our investigation to marry a CPS with a parallel capable infrastructure, like the SCC, that can provide the necessary knobs to tradeoff power and energy restrictions with application runtime deadlines.

For our evaluations, we generate custom settings for the SCC. These settings initialize the SCC with different core, network and memory configurations. Because we envision multiple programs communicating at the same time on the SCC, we also studied non-standard mesh network speeds in the hope that it could provide us with additional insights on the tradeoff space of the SCC.

Both ROMS and the path planning application make use of RCKMPI [10] for message passing which should provide comparable performance to RCCE [10], [11]. The MPD process manager, recommended for use with the SCC, is used throughout the experiments. All cores boot a Linux 3.1.4 kernel image. Finally, power measurements were gathered from the SCC infrastructure.

A. ROMS benchmark

We evaluated the feasibility of running ROMS on the SCC using a sample benchmark provided with ROMS. The benchmark consists of 512x64x30 grid points and 200 time step iterations. The main computation is a two-dimensional stencil with nearest-neighbor communication. The grid is divided into tiles, where the total number of tiles must match the number of cores that are part of the computation. The grid's tile dimensions were chosen to maximize the size of grid points calculated per core and to reduce the size of the halo/ghost regions. Larger halo regions require more communication and computation. We empirically validated that the tile dimensions used are optimal for the grid size and the number of cores.

Our evaluation of the ROMS benchmark program is shown in Fig. 2. A diverse set of configurations of CPU, mesh, and memory were tested with both 24 and 48 cores. In most cases, the runtime for 48 cores, Fig. 2(b), is lower than the 24 cores (Fig. 2(a)) with the same setting. The fastest configuration with 24 nodes performed nearly identically to the second slowest configuration of 48 nodes, and outperformed the slowest. In scenarios where soft runtime deadlines are acceptable, numerous options and tradeoff points are available for ROMS. A global application scheduler can consider these alternatives during the arbitration of the next SCC setting.

The average power consumption during the execution of ROMS is shown in Fig. 2(c) and Fig. 2(d) for 24 and 48 nodes, respectively. Throughout the experiments, lower mesh speeds reduced power by several watts. The most pronounced effect on power were high tile frequencies. Battery operated CPSs, like the Slocum glider, may need to observe power caps during operation, since actuators and other systems can increase the power load on the device. Therefore, it may not always be an option to run the fastest configuration with the highest node count.

Similar to the runtime, it is generally more energy efficient to use 48 cores instead of 24 cores to run the benchmark. The highest setting for the 24 nodes in Fig. 2(e) is, however, similar to the lowest configuration of the 48 cores seen in Fig. 2(f). When comparing their respective runtimes, the 48 node setting does outperform the 24. Across the figures, the crossover points are very similar and are prospective tradeoffs opportunities. Because of the dynamic nature of AUVs, mission priorities can change often, emphasizing the importance of a suitable arrangement for runtime, power and energy.

B. Path Planning

We have ported both the serial (S-TVE) and parallel (P-TVE) versions of the TVE path planning algorithm to the SCC. The input parameters to both programs were identical throughout the benchmark tests. Since parameter choice can have an impact on the amount of parallelism the program is capable of during execution, we have chosen a set of parameters consistent with our previous work [12].

The opportunity for parallelism that was exploited and implemented in P-TVE was to find the optimal dive profile depths for the vehicle. Because the AUV can experience different currents at various depths, it may be advantageous for the vehicle to glide within a certain depth range for portions of the flight. For each edge in the graph, this dive profile calculation is evaluated for 20 distinct depths ranges.

Results of the path planning programs for the SCC configurations are show in Fig. 3. S-TVE results are only available for one core since there is no parallelism involved. P-TVE has a master/slave architecture where the master delegates work to slaves that perform the dive profile task, so at least two cores are required. The MPI-NOOP results measure the overhead of the MPI infrastructure. It is a modified version of P-TVE which initializes MPI and immediately exits.

The program runtime, Fig. 3(a), and dive profile search time, Fig. 3(b), decreased as the number of cores increased for P-TVE. There is an initial communication overhead for two cores, when compared to S-TVE, as the master must delegate work to the slave. The step-wise behavior is explained by the number of iterations of work delegation that is performed by the master. For example, with 11 cores, 10 slaves perform work for two work iterations. In the case of 12 cores (11 slaves), the second work delegation will leave one slave idle. Because of the input parameter of 20 distinct depth range calculations, the optimal number of nodes should be 21. This accounts for one master with 20 slaves doing one iteration of work. Additional nodes only provide overhead in P-TVE as indicated by the

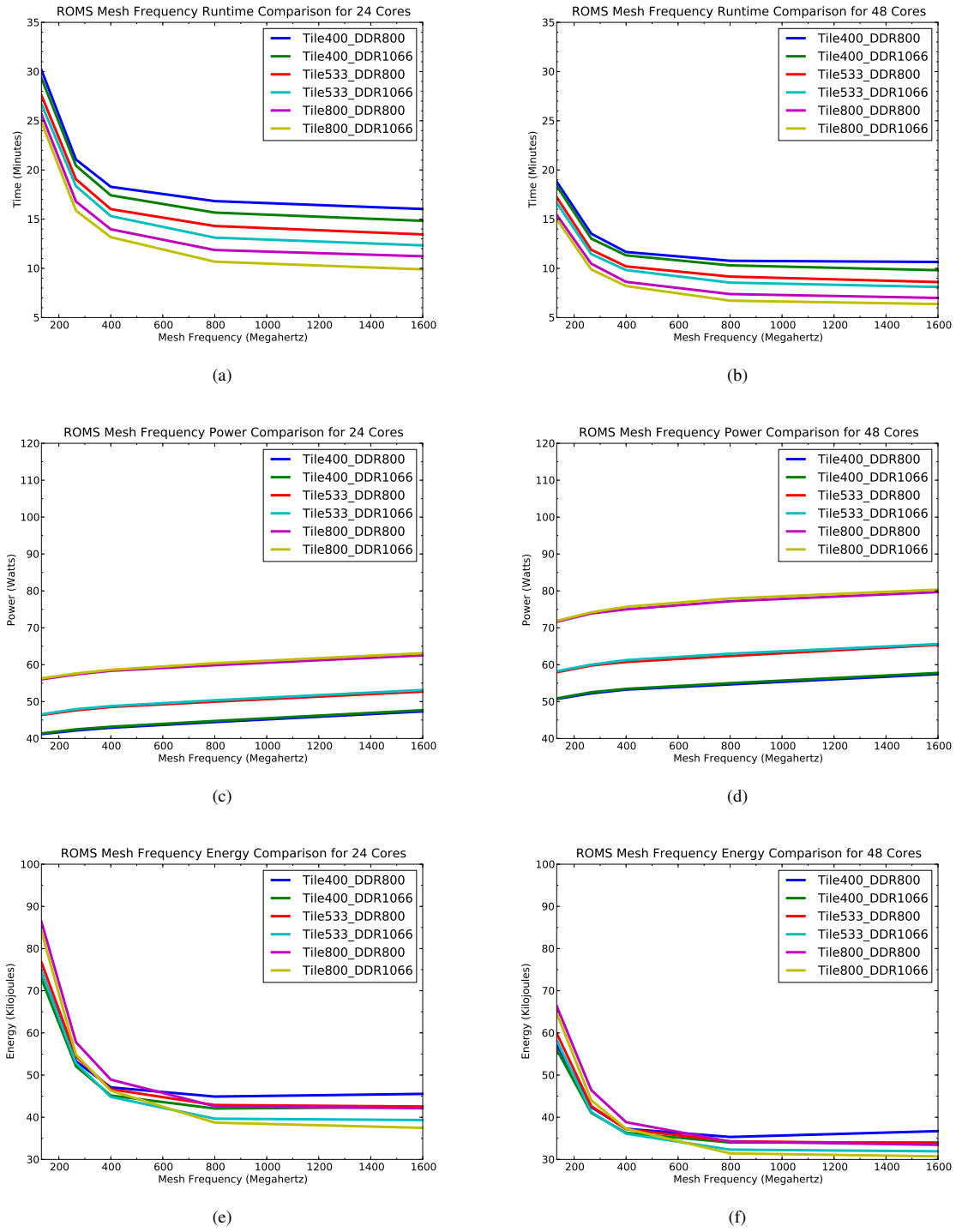


Fig. 2. ROMS evaluation results for various SCC settings. The execution times for ROMS using 24 (a) cores and 48 (b) cores. The average power , (c) and (d), of the SCC during the execution of program. The energy required to run ROMS for 24 (e) and 48 (f) cores.

speedup of the dive profile search in Fig. 3(c). The speedup for each setting is normalized to the S-TVE search time of the same setting. If the number of profile searches is increased, additional cores could be used with a concomitant increase in benefit. Additional details are available in [12].

To reduce the power and energy of the program, idle slaves are instructed by the master to enter into sleep mode. In sleep mode, a slave performs an asynchronous receive call instead of a blocking receive call. This allows the slave to sleep in

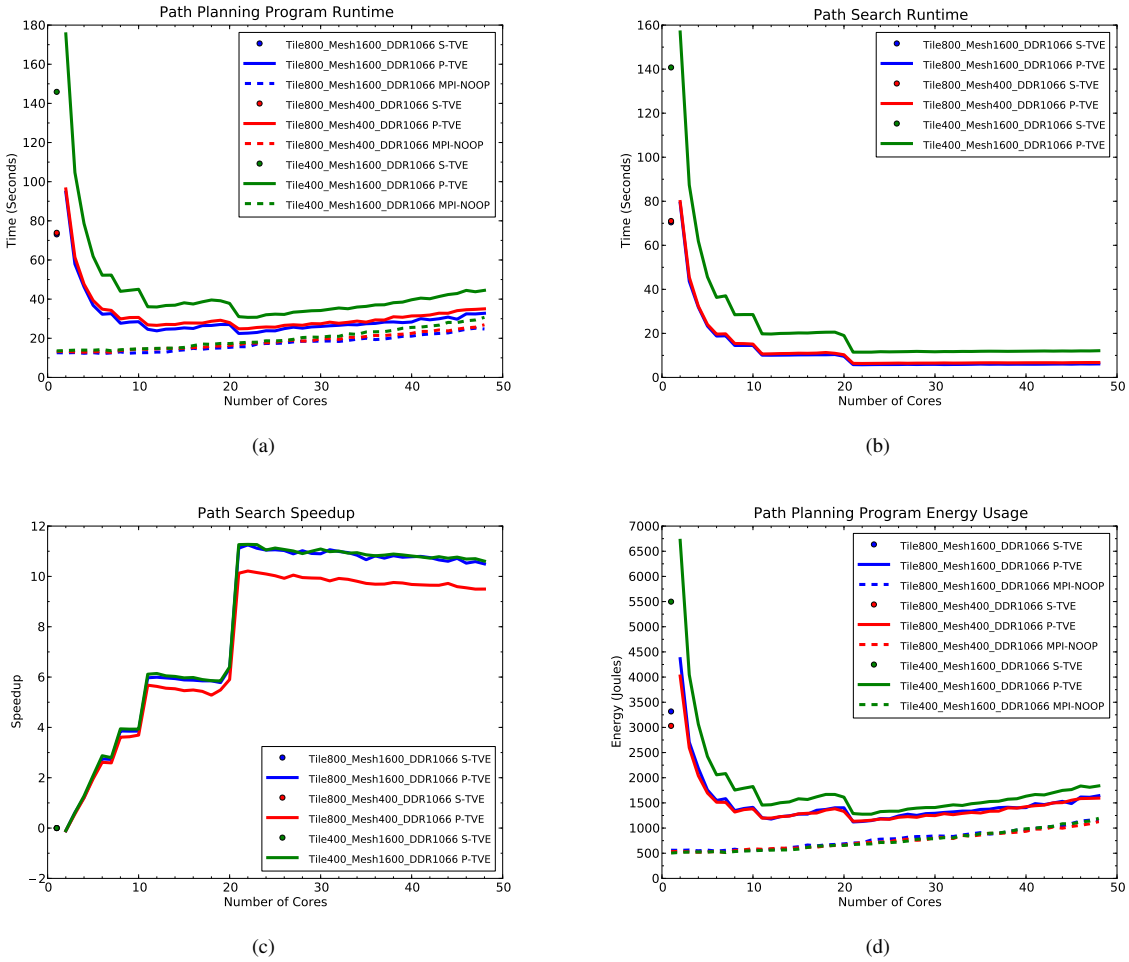


Fig. 3. Evaluation results for the path planning program for various SCC settings. The runtime of (a) is the time required for the entire program to execute. The search time, (b) is the time required to perform the search for the optimal dive profile. Speedups for each of P-TVE is relative to the S-TVE with the same SCC setting. The energy required for the entire program execution (d) is based on (a).

between update checks of the asynchronous call. Although this introduces latency for the first receive, it greatly reduces the overall energy used by the slave. This latency is evident in Fig. 3(c), especially when there are a high number of idle slaves. For example, after 21 nodes, even the idle slaves that will never perform any work experience the latency because they wait for the termination message to be sent by the master.

The evaluation indicates that the path planning program is more reliant on computation than communication as the slowest core speed setting has the longest program and profile search runtimes. Lowering the mesh speed does decrease the speedup of the parallelization because it delays communication between the master and its slaves. However, the effect it has on runtime is not as significant as observed when changing the CPU frequency.

The energy required for the planning programs are depicted in Fig. 3(d); it shows opportunities for tradeoffs that could be used when choosing an SCC configuration that will run several programs simultaneously on the chip. Although the runtime of the P-TVE is generally longer for the low mesh speed configuration, the power saved by reducing the mesh frequency translates to a comparable energy profile of the highest speed SCC setting. After 21 cores, even the slowest tile setting could be considered, as the energy difference is not substantial. Similar to the runtime results, energy is wasted on idle slave cores. We hope to address this issue in the future.

C. Discussion

The applications described, along with others, could be required to run simultaneously on the SCC. Depending on the current needs of the system the priority of tasks may change periodically, or change based on observations of phenomena in the environment.

Power caps can also restrict the selection of high power SCC settings. A Slocum glider typically uses alkaline battery packs, so the supply voltage drops as energy is consumed. A glider's fresh alkaline battery pack is rated as 1800 watt hours, while the SCC's power demands can range from 40 W to 80 W for

our applications. As a comparison, the buoyancy engine of the glider operates at 60 W or more during inflections at 200 m depths. The vehicle must maintain a minimum voltage level at all times to operate safely. The use of actuators, like the buoyancy engine, and sensors, such as a DVL, will increase the power needed by the AUV. It may not be possible to run the SCC concurrently with some sensors, while other sensors can be active at the same time as the SCC provided that the chip does not exceed its allotted power.

Having knowledge of the tradeoff points for an application is critical when choosing a configuration setting. For example, at some point in a deployment, a vehicle's path may need to be resolved rather quickly. Ideally, the highest tile, mesh and memory speed (Tile800_Mesh1600_DDR1066) should be chosen and P-TVE is run on 21 nodes. However, there could be a loose deadline to perform modeling and thus ROMS must also be considered. If the highest setting exceeds the allotted power, a small sacrifice could be made by lowering the mesh frequency. The impact on the runtime and energy of path planning is minimal. While the impact is greater for ROMS, it may still fall within the soft deadline restrictions.

The ROMS tradeoff scenario described in Section IV-A could also be made in the case of a power cap. If there is no need for path planning, and the requirements are such that ROMS should have nearly the same runtime and energy profiles as the best setting for 24 nodes, then the program could be run on all 48 cores at half the tile frequency. This allows the program to not only be more runtime and energy efficient but also greatly reduces the required power. The lowering of the frequency, in this case, is what may be needed to bring the power profile below the cap.

Although we have focused on power cap scenarios, other tradeoff points which concentrate on energy and runtime can be made. This is especially true if more applications, like sensor triggering, are involved in the deliberation. Other cyber-physical systems will have their own hardware and software restrictions and priorities. The SCC can provide CPSs a tradeoff space in which it can make decisions that involve runtime, power, and energy.

V. CONCLUSION AND FUTURE WORK

In this paper, we have performed a set of benchmarks for applications on the SCC that could be used in a battery-operated Slocum glider or a battery-operated propeller-driven AUV. The results of our evaluation indicated that the applications expose many knobs for different SCC tile, mesh and memory frequency settings. These knobs can be used to tradeoff program runtime, power, and energy use depending the needs of the AUV and the overall mission. A deployment of our SCC system within one of our Slocum gliders is not possible due to the SCC's particular form factor and system configuration. Our case study shows that future many-core architectures similar to the SCC can play a significant role in making AUVs more effective, autonomous research platforms.

As part of future work, we would like to port our applications to invasive MPI (iMPI) [13]. In particular, based on our evaluation of the path planning application, the overhead of launching MPI processes can be significant using MPD. Similar overheads were observed in [13] and were reduced using iMPI's process manager. We hope that this MPI alternative will help to reduce the energy requirements of applications running on the SCC.

We would also like to extend our evaluation with additional SCC settings. The programs in our evaluation were run in isolation. Performing a similar analysis for multiple applications running concurrently on the SCC could be of interest. Having several programs interacting with the chip may spur interesting effects on the runtime and energy profile of the applications.

ACKNOWLEDGMENTS

This research has been partially funded by NSF awards CSR-CSI #0720836 and MRI #0821607, the Murray Visiting Professorship at Rutgers University, and the Internal Excellence Promotion for Research at Ilmenau University of Technology. We are particularly grateful to Intel and the SCC development team for their help and advice throughout this project.

REFERENCES

[1] A. Peter Greenhalgh, "Big.little processing with ARM cortex -a15 and cortex-a7 www.arm.com/files/downloads/big.little_final.pdf," 2011.

[2] M. Gries, U. Hoffmann, M. Konow, and M. Riepen, "SCC: A flexible architecture for many-core platform research," *Computing in Science Engineering*, vol. 13, no. 6, pp. 79 –83, November-December 2011.

[3] Teledyne Webb Research, "Slocum glider," Falmouth, MA, http//www.webbresearch.com/slocum.htm.

[4] J. G. Graver, R. Bachmayer, N. E. Leonard, and D. M. Fratantoni, "Underwater glider model parameter identification," in *Proceedings 13th International Symposium on Unmanned Untethered Submersible Technology*, 2003.

[5] Persistor Instruments Inc., "Cf1 computer system," Marstons Mills, MA, http://www.persistor.com.

[6] H. Woithe and U. Kremer, "A programming architecture for smart autonomous underwater vehicles," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2009. IROS 2009.*, October 2009.

[7] A. Shchepetkin and J. McWilliams, "The regional oceanic modeling system (ROMS): a split-explicit, free-surface, topography-following-coordinate oceanic model," in *Ocean Modelling*, vol. 9, 2005, pp. 347–404.

[8] M. Eichhorn, "A new concept for an obstacle avoidance system for the AUV "Slocum glider" operation under ice," in *Oceans '09 IEEE Bremen*, Bremen, Germany, 11-14 Mai 2009 2009.

[9] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, no. 1, pp. 269–271, 1959.

[10] I. A. C. Ureña, M. Riepen, and M. Konow, "Rckmpi - lightweight MPI implementation for Intel's single-chip cloud computer (SCC)," in *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, ser. EuroMPI'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 208–217.

[11] T. Mattson, R. Van der Wijngaart, 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 *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, November 2010, pp. 1 –11.

[12] M. Eichhorn, H. C. Woithe, and U. Kremer, "Parallelization of path planning algorithms for AUVs concepts, opportunities, and program-technical implementation," in *Oceans '12 MTS/IEEE Yeosu*, Yeosu, Republic of Korea, May 21-24, 2012.

[13] I. A. C. Ureña, M. Riepen, M. Konow, and M. Gerndt, "Invasive MPI on Intels single-chip cloud computer," in *Architecture of Computing Systems ARCS 2012*, ser. Lecture Notes in Computer Science, A. Herkersdorf, K. Rmer, and U. Brinkschulte, Eds. Springer Berlin / Heidelberg, 2012, vol. 7179, pp. 74–85.

 ISBN: 978-2-7257-0016-8

Interactive Visualization and Task Management on the 48-core Intel SCC

Jimi van der Woning and Roy Bakker
Informatics Institute, University of Amsterdam
Sciencepark 904, 1098 XH Amsterdam, The Netherlands

***Abstract*—In this paper we propose and describe how we have built a tool that enables a user to interactively monitor and manage a many-core system like the 48-core experimental Single-chip Cloud Computer (SCC), which was created by Intel Labs targeting the many-core research community. We provide the user with a visual representation of the current state of the system on multiple levels of detail, such as chip, core and task. We allow the user to create, start, pause and migrate tasks across different cores. We also allow the user to easily adjust the voltage and frequency of the chip. However this tool can run on any PC with a screen and input devices, we have optimized the interface to run on a multi-touch device for the best ease of use.**

I. INTRODUCTION

The Single-chip Cloud Computer (SCC) experimental processor [1] is a 48-core *concept vehicle* created by Intel Labs as a platform for many-core software research. It provides an on-chip message passing network, a non cache-coherent off-chip shared memory and dynamic frequency and voltage scaling.

Unlike currently available multi-core systems, the SCC is an on-chip distributed system. Even though efforts are already made and still continuing on writing and porting operating systems or virtualization layers that can manage the chip as a whole [2], the most common use of the SCC system is currently to have every core managed by its own instance of a slightly modified version of the Linux kernel. As a consequence, it gets harder for users to gain insight in the current state of the system. Also, it is not trivial to map tasks on the system while keeping the load balanced and energy consumption low, without a complete understanding of the state of the system.

We propose a management system for independent many-core systems like the SCC, which enables users to interact with the system. The user of the system must be able to:

1) Monitor the system:
   - Current load, state and power consumption of the chip.
   - The current resource usage for each core.
   - The resource usage per monitored process on a core.
   - Task output.
   - Overview of running, waiting, completed and possibly failed tasks.
2) Manage the system:
   - Easily create a task.
   - Start a task on a single specified core, or the best core available (suggested by the system).
   - Migrate tasks to other cores, either manually with or without suggestions from the system, or potentially automatically.
   - Control frequency and voltage.

All of the above can easily be controlled from a user interface running on a regular PC with mouse interaction, but to improve the user experience and ease of use even more, the system is optimized for use with a multi-touch system.

Manually managing a system like the SCC does not seem to be very efficient in daily practice. For this purpose, it is better to use an automated grid- and cluster management system. However, we think that our system is very useful in both research and education. It can mainly be used for experimenting with task placement and voltage and frequency settings, while having a clear understanding of what is currently happening from a user friendly interface. The system can later be attached to an automated task manager or grid engine.

In this paper we will discuss our experiences in building such a management system for the SCC. We assume sufficient knowledge of the SCC architecture and its memory system, as this is broadly covered by both related work [1], [3] as well as our previous work [4], [5]. Some related work on visualization, but not on cluster management systems, is described in Section II. We discuss which approaches and tools we have used for the implementation in Section III. In Section IV we evaluate these different approaches and we conclude with a discussion and future work in Section V.

II. RELATED WORK

Since the beginning of the SCC research, a *Performance Meter* has been available as part of the *sccKit* provided by Intel (See Figure 1). This tool shows us the current load for all the independent cores, as well as the overall usage and power consumption of the system. It is a nice tool, but it will not be sufficient for our proposed system, as there is no way to actually manage the SCC.

QNX Software Systems [6] has developed software for the visualization of many-core applications, which is used mostly for the development of applications in embedded systems. It is not considered for the management of a running system. Other visualization frameworks such as IBM's Tuning Fork [7] are available to monitor the performance of a system. Many visualization tools only operate on traces instead of a running system. We have not found a system that combines visualization and management on the same high level as we want to.

Figure 1: The SCC performance meter (`sccPerf`).

III. MANYMAN - THE MANY-CORE MANAGER

The many-core visualization and management tool has been dubbed *ManyMan*, as in Many-core Manager. ManyMan consists of two main parts, a front- and a back-end, with a communication layer in-between. The reason for this separation is twofold. First, the front-end could theoretically be attached to a different many-core chip, or, the other way around, a different front-end could be attached to this back-end. This increases the usability of the tool, since it is not restricted to just one chip or interface. The second reason for this separation is of a more practical kind. At the University of Amsterdam, the SCC is located in a server room where no monitor could be easily attached to it.

BACK-END

The back-end has been written completely in Python, for which there are two main reasons. First, Python is a relatively easy programming language which allows for rapid development. Second, Python provides good support for running many threads, starting shells on remote machines using subprocesses and TCP communication. The back-end performs multiple tasks:

A. Monitoring

Monitoring the chip is half of the visualization and management tool. One would like to know the status and payload of every core and task on the system. Unfortunately, the SCC does not provide such information about the chip as a whole, which means that it needs to be retrieved separately from each core.

In order to access one of the SCC's cores, one needs to open an SSH connection to the core in question from the MCPC. Since an SSH connection needs to be opened for each task that runs on the SCC, SSH Connection Sharing is used. This mechanism allows us to have only one TCP connection to an SCC core, with one-time authentication. This master connection has to be active while all subsequent connections are, which makes the monitoring process the perfect candidate to be that connection.

In order to obtain the payload information of each core, the Unix `top` command is used. At adjustable intervals, `top` provides information about all processes that are running on the core and the total payload of the core itself. The fact that this information is everything that has to be shown, makes `top` the ideal monitoring solution. However, as `top` accesses more information than just that information that is needed, it might create some overhead.

It has to be noted that any resource usage of processes that have not been started using ManyMan, as for example kernel processes, will be marked as overhead. This overhead will be visible in the total core payload, but, of course, not in the per task payload. Because of that, the task payloads will not add up to the total core payload.

B. Task creation

When a task needs to be created, a child connection is added to the monitoring master SSH connection of the core on which the user has decided the task should run. On this child connection, the task is started with BLCR's `cr_run` command, and a small wrapper that enables us to obtain the process identifier (PID) of the task on the remote core. When the program starts to run, its output will be buffered. It will be sent to the front-end upon request.

In case a user does not know which core to start a task on, a *smart-start* function has been implemented. When smart-starting a task, the core with the least CPU and memory usage is selected. In this process, both the CPU and the memory usage have the same weight. A possible growth in CPU or memory usage is foreseen by also taking the number of running tasks on a core into account. The more tasks are running on a core, the smaller the chance a task will start there gets. As soon as the best core to run the task on is found, the task will be started on that core as usual. Note that this smart-start function does not keep track of the history of core usage or whatsoever, but it just looks at the current core state.

C. Task migration

To enable the migration of tasks, we make use of the Berkely Lab Checkpoint/Restart library [8]. Using BLCR, tasks can easily be stopped (checkpointed) and restarted later while releasing all resources. Restarting a task can also be done on other cores or even other compatible machines. Besides the benefits, checkpointing also creates overhead, as the BLCR library writes the complete state of the process to the filesystem. A task that needs to be migrated will first be checkpointed using the `cr_checkpoint` command from the BLCR library. The location of the context file that is hereby created will be stored in order to be able to restart the task later. When checkpointing is complete, the task can be restarted on

the desired core using `cr_restart`. When a user does not want to restart the task yet, it will instead be moved to the list of waiting tasks. Since all cores of the SCC mount the same `/shared` directory, one does not have to worry about sending context files among cores. These can just be found on the exact same path as where they were originally stored. At restart, the `cr_restart` command will be executed with the `-no-restore-pid` flag to avoid PID collisions on the remote core.

Just like the smart-start function, a *smart-move* function has been implemented. Using this function, a task can be moved to the best possible core, which is found the same way as it is done when smart-starting, except for the fact that the new core will never be the core the task is already running on.

D. Task pausing / resuming

Since checkpointing a task produces some overhead, tasks can also be paused using the traditional POSIX STOP signal, after which they can be resumed by sending the POSIX CONT signal. It has to be noted that even though the process is paused and will not use the CPU, it will not release resources such as memory. Due to this fact, paused tasks cannot be moved to other cores, as long as they are not checkpointed. Besides that, not releasing memory might be a problem for the SCC cores, since their private memory is limited (around 640MB). For manually scheduling CPU intensive tasks however, this is a great solution.

E. Communication

The communication between front- and back-end makes use of a TCP connection. The connection is currently open and not encrypted as both client and server are within the same access restricted network. When the server needs to be available on the public internet, some form of authentication has to be implemented. Across the TCP connection, messages are sent in the JSON format. This format is human-readable, which allows for easy debugging and portability.

FRONT-END

The GUI part has been optimized for use with a (multi)touch display. For the front-end we make use of *Kivy* [9]. Kivy is an open source library for rapid development of applications that make use of innovative user interfaces, such as multi-touch applications. The same Kivy source code runs on Linux, Windows, MacOSX, Android and IOS, providing us with the best flexibility.

F. Chip overview

The chip overview as a whole is shown in figure 2. In the middle part, the cores are presented in the order they are physically arranged at on the chip. This order serves no functional purpose, but has been chosen to provide the user a realistic view of the chip and provide the same layout as the `sccGui` does. The payload of each core is visualized by a coloured overlay that changes in both colour and size. The portion of the core that is taken by the overlay literally

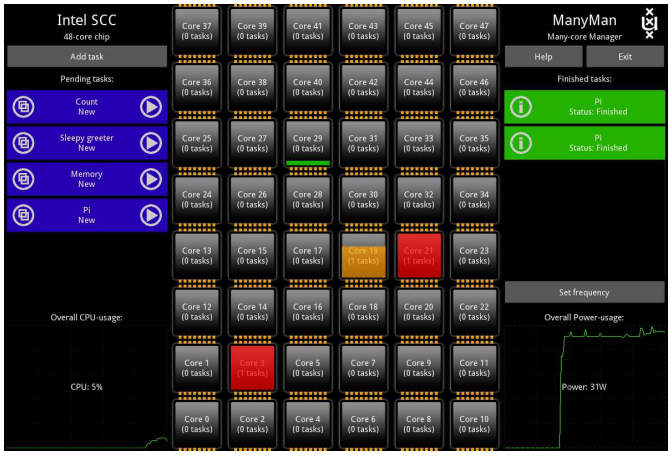


Figure 2: ManyMan's main window, the chip overview.

translates to the core's CPU usage, where the colour changes from green at 0% CPU to red at 100% CPU. Although the information is only updated once a second, the overlay is animated to fade to the new payload within that second. The main window also notes the number of tasks per core.

On the left side of the window, a list of tasks is shown. These tasks are currently not running on any core, but are either not started yet (*new*) or have been stopped by the user (*stopped*). In order to (re)start such a task, one can simply drag it to the core he or she wants it to run at. If the user does not care on which core the task will run, the task may be *smart-started* by tapping the play button on the right side of the task. By clicking the copy button on the left side of a task, the task can be duplicated. This is especially useful when said task is a benchmark program.

The right side of the window provides a list with tasks that are either finished or have failed to complete. In this list, the task's output and statistics can still be viewed, but it can not be dragged any more. The delete button will remove the task completely from the ManyMan system.

Figure 3: ManyMan's core view, containing information about core 28.

G. Core view

When tapping a core in the chip overview, a popup like the one in figure 3 will open. On the left side, the history of the CPU and memory usage is shown. In these graphs, the white line indicates the total load of the core, which consists of all tasks started by ManyMan, plus all overhead (i.e tasks not started through ManyMan, and OS overhead). The coloured lines in the performance graphs indicate the payload of the tasks that have been started using the many-core management system. The colours of these lines match the colours of the tasks in the task list on the right side of the core view.

The tasks in the task list can be moved to a different core by simply dragging them to the core a user wants them to run on. When dragging a task, all open core popups will swerve out of the way so that they do not block any core. The *smart-move* option, along with some additional controls and information, is located in the detailed task view. This task view can be opened by tapping the information button on the left side of a task. When a task is dragged to anything that is not a core, as for example the task list on the side of the main view, it will be checkpointed and moved to the chip overview's task list.

In order to be able to compare the payloads of two or more cores, multiple core views can be opened at once. The user may drag them around to prevent them from lying on top of each other. The popup can also be scaled to fit more of them on the screen, or rotated for when the user wants to look at it from a different angle.

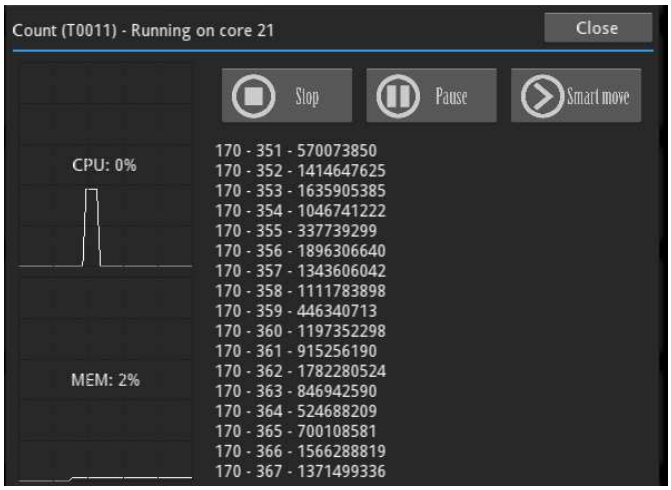


Figure 4: ManyMan's task view, on a task 'Count'.

H. Task view

The detailed task view (see figure 4) looks similar to the core view. Again, the left side of the popup contains information about the CPU and memory usage of the task. On the lower right side of the window, the last 100 lines of the task's output are shown. This number is configurable, but cannot be too large due to Kivy's inefficient way of rendering text. The complete output of a task is written to a file, so that it can be accessed and processed later.

Above the output, the task control buttons are shown. Tapping the stop button will signal the back-end that the task needs to be checkpointed, after which the task will be moved to the chip overview's task list. When a user wants to temporarily pause a task, he can tap the pause button. The back-end will then send a POSIX STOP signal to the task, after which the pause button will be replaced by a resume button. Tapping this button will cause the task to be resumed by sending it the POSIX CONT signal. Finally, tapping the move button will *smart-move* the task to the best available core other than itself.

Figure 5: The task create popup along with the on-screen keyboard.

Just like the core views, multiple task views may be opened at once to compare their performance. It is even possible to have both multiple core and task views open at the same time.

I. Task creation

When tapping the `Add task` button in the main window, a popup will open in which a command and optionally a name can be entered. This popup is shown in figure 5. For multi-touch support, an on-screen keyboard can be used to enter the name and command of the task. A command is either a known shell command or the location of a binary accessible on the cores (for example in the `/shared` directory). After the create button is pressed, the task will be added to the chip overview's task list. A better way of creating a task would be by selecting a binary using a file browser. Unfortunately, the front-end does not run on the SCC's MCPC, which means that one cannot easily open a graphical file browser and navigate to the file.

J. Voltage and Frequency Scaling

Using the `Set Frequency` button, ManyMan is able to set the frequency divider for each tile. Currently, we only change frequencies at voltage domain level, and set the voltage according to the minimal value for that frequency. See table I, obtained from [10], for the corresponding values. As figure 6 shows, one can set the frequency for each of the power domains, or for the chip as a whole. In the main window, the power consumption of the complete chip is visualized in a graph.

Figure 6: Frequency and Voltage settings, easily adjustable.

Frequency divider	Frequency	Voltage
2	800	1.16250
3	533	0.85625
4	400	0.75625
5	320	0.69375
6	267	0.66875
7	229	0.65625
8	200	0.65625
16	100	0.65625

Table I: SCC supported frequencies and required voltage. Based on [10], where frequencies between 100-200 are left out on purpose.

IV. EVALUATION

A. Running latency

In order to be able to stop a task in mid-execution, tasks are started using the `cr_run` command, which might create overhead. In order to test this, experiments have been performed using a program that calculates the sum of some million random numbers. The Unix `time` function has been used to time this program when it is executed both with and without the `cr_run` command. On the SCC, the `time` function is unreliable for measuring the wall-clock time, due to the possibly varying frequencies of the cores where the kernel does not correct for by default. In a period that the frequency does not change, however, this function can be used to measure relative times. We measured an overhead of about 0.2% for this task with a running time of about 55 seconds. In practice, it boils down to less than 0.1 second of overhead per execution.

B. Checkpoint/Restart latency

Checkpointing can create a lot of overhead, as the BLCR library writes the complete state of the process to the filesystem. On the SCC, the only (persistent) filesystem is NFS mounted from the MCPC. This allows easy migration of tasks across cores, but also introduces a large latency for task migration.

Figure 7 shows the latency for checkpointing and restarting processes with increasing memory usage. The testing program

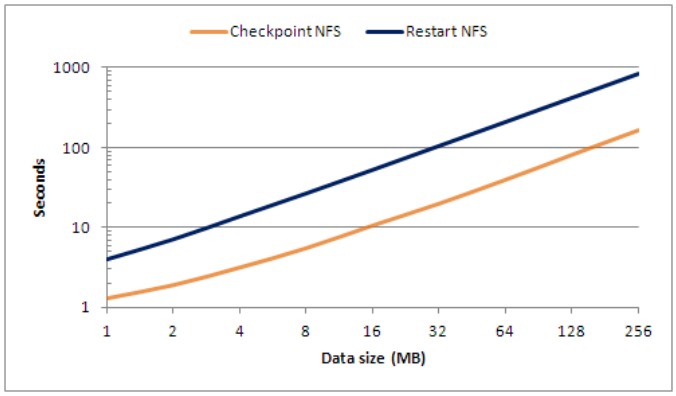


Figure 7: Latency for checkpointing and restarting using the BLCR library on SCC core 0.

that has been used here simply allocates a specified number of megabytes of memory. This memory is then filled with random data, after which the sum of this data is calculated. While calculating the sum, the process will be checkpointed. It is made sure that all requested memory has been allocated and filled with random data at that time. This way, problems with *lazy allocation* of memory pages will not be encountered.

The measurements in this experiment were done by adding a timing mechanism in the BLCR source code. At the beginning and end of the main function, the time stamp counter (TSC) is read and the values are subtracted. To convert to seconds, the resulting value is divided by the core frequency. As expected, the time required for checkpointing scales linearly with the amount of allocated memory.

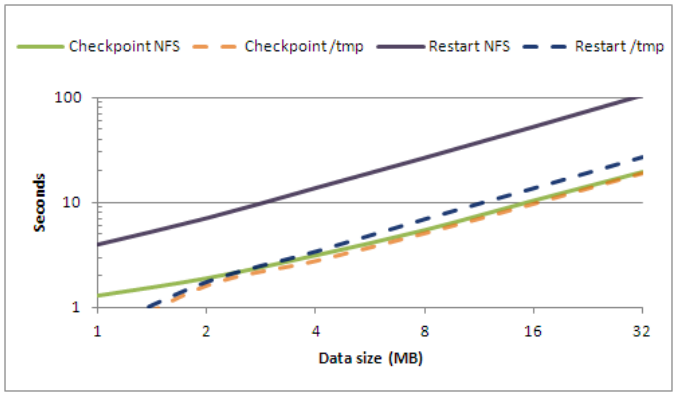


Figure 8: Latency for checkpointing and restarting using the BLCR library on core 0. Writing to NFS compared to writing to RAM.

Besides the test where context files were written to NFS on the MCPC, an additional test has been performed where the context files were written to the filesystem in RAM (`/tmp`). The results of this experiment can be found in figure 8. It shows that the time required for checkpointing can be reduced with approximately 4.5%. When restarting a task from RAM, the difference is much bigger. A task now restarts almost twice as fast, with a speedup of 49.3%. This giant difference is due to the fact that loading data from NFS into RAM is a very time consuming task. When context files are written in RAM,

there is a problem when tasks need to be migrated. As multiple cores do not share their RAM, the context files would have to be copied between cores. This could possibly be done by using *Copy Cores* or shared memory (memory remapping). However, relocating context files would slow down and complicate the restarting process again and probably not be beneficial.

C. Connection Sharing time gain

In order to speed up each access to a core, SSH Connection Sharing is used. We measured the average time it takes to open an SSH connection to SCC core 10, both with and without using Connection Sharing. In order to obtain these results, an `ssh` command was used to open a connection to core 10, on which immediately the `exit` command was executed. We measured an average speedup of 0.61 seconds when using Connection Sharing.

In order to make sure these results were not distorted due to immediately closing the connection using the `exit` command, additional experiments have been performed in which a `sleep` of 10 and 20 seconds has been executed. For these experiments, the average time gains were 0.62 and 0.61 seconds respectively, by which the initial measurement is confirmed.

D. Energy Consumption

We performed a very small power consumption measurement in which we calculate the number of (floating point) operations in an iterative estimation of π. The results of this test can be found in table II. We can go from as low as 21 W at 100 MHz to 110 W at 800 MHz. As we count the number of operations per watt, we see that 320-400 MHz is the most efficient in power consumption. For an idle system, we can easily scale back to 100 MHz and only consume 18 Watts.

Freq.	Volt.	FLOP/s	Power	FLOPs / Watt	Idle
800	1.16250	2232382092	110W	20294383	64W
533	0.85625	1517496998	48W	31614521	30W
400	0.75625	1138837303	35W	32538209	24W
320	0.69375	912457180	28W	32587756	22W
267	0.66875	760936186	25W	30437447	21W
229	0.65625	653551028	24W	27231293	20W
200	0.65625	570407521	23W	24800327	19W
100	0.65625	285427629	21W	13591792	18W

Table II: SCC power consumption

E. Usability test

In order to test the usability of the front-end, a few Computer Science students and a couple of students from non-computer related disciplines were asked to perform a number of tasks. After these tasks had been completed, the students were asked a number of questions about the usability of the software. The tasks that had to be performed and the questions that have been asked can be found in [11], together with a more detailed analysis of the results.

The general opinion of both the Computer Science and the non-Computer Science students was that the tool looked great. They all found the way tasks have to be started very intuitive and really liked the detailed core overview. During these tests, some remarks were made by the participants of which most have been added to the application.

V. CONCLUSION

The proposed application provides any user (either a computer scientist or not) with a total insight of the resource usage on a many-core system like the SCC. The current design of the system is modular, which means that we can easily adapt the font-end to work with a different many-core system than the SCC, but we can also use the back-end for other purposes. The current system may also be usable to manage a cluster of independent (unix) machines with a shared file system, as we currently consider the SCC as a cluster on chip.

The work described in this paper can be extended with some more additional features. Examples of those can be support for MPI tasks including core selection, or automated task scheduling and migration. This scheduler should then be able to automatically adjust voltage and frequency based on the load of the system and possible deadlines. One can also investigate the options for replacing the BLCR library with a more sophisticated cluster management system.

The software created in this project is available for download under the GPL3 license at [12], where we also provide more information about the project, screenshots and a demonstration video.

REFERENCES

[1] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pp. 108–109, February 2010.

[2] S. Peter, A. Schüpbach, D. Menzi, and T. Roscoe, "Early experience with the Barrelfish OS and the Single-chip Cloud Computer," in *3rd Many-core Applications Research Community (MARC) Symposium*, KIT Scientific Publishing, September 2011.

[3] Intel Labs, *SCC External Architecture Specification*, revision 1.1 ed., November 2010.

[4] 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 *3rd Many-core Applications Research Community (MARC) Symposium*, KIT Scientific Publishing, September 2011.

[5] R. Bakker and M. W. van Tol, "Experiences in porting the SVP concurrency model to the 48-core Intel SCC using dedicated copy cores," in *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium* (P. Tröger and A. Polze, eds.), no. 55, Feb. 2012.

[6] "Qnx introduces breakthrough in multi-core visualization tools (http://www.qnx.com/news/pr_2094_1.html)," 2006.

[7] D. Bacon, P. Cheng, D. Frampton, D. Grove, M. Hauswirth, and V. Rajan, "Demonstration: On-line visualization and analysis of real-time systems with tuningfork," in *Compiler Construction* (A. Mycroft and A. Zeller, eds.), vol. 3923 of *Lecture Notes in Computer Science*, pp. 96–100, Springer Berlin / Heidelberg, 2006.

[8] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 494, 2006.

[9] "Kivy (http://kivy.org)," 2012.

[10] P. Gschwandtner, T. Fahringer, and R. Prodan, "Performance analysis and benchmarking of the intel scc," *Cluster Computing, IEEE International Conference on*, vol. 0, pp. 139–149, 2011.

[11] J. van der Woning, "Interactive visualization and dynamic task management of many-core systems. A case-study: The Intel Single-chip Cloud Computer," bachelor's thesis, University of Amsterdam, jun 2012.

[12] J. van der Woning, "ManyMan." Online, http://student.science.uva.nl/~jimivdw/manyman/ [visited jun 2012].

 ISBN: 978-2-7257-0016-8

Modelling Power Consumption of the Intel SCC

Patrick Cichowski*, Jörg Keller* and Christoph Kessler†
*Faculty of Mathematics and Computer Science
FernUniversität in Hagen, 58084 Hagen, Germany
Email: {patrick.cichowski,joerg.keller}@fernuni-hagen.de
†Dept. Computer and Information Science (IDA)
Linköpings Universitet, 58183 Linköping, Sweden
Email: christoph.kessler@liu.se

***Abstract*—The Intel SCC manycore processor supports energy-efficient computing by dynamic voltage and frequency scaling of cores on a fine-grained level. In order to enable the use of that feature in application-level energy optimizations, we report on experiments to measure power consumption in different situations. We process those measurements by a least-squares error analysis to derive the parameters of popular models for power consumption which are used on an algorithmic level. Thus, we provide a link between the worlds of hardware and high-level algorithmics.**

I. INTRODUCTION

Energy consumption by computers is continuously growing, and thus energy-efficient computing — sometimes embellished as the wider field covered by the buzzword *green computing* — gains more and more interest. The Intel Single-chip Cloud Computer (SCC) manycore processor chip supports energy efficiency by allowing the user to scale the clock frequency and the supply voltage of the cores (in groups of 2 and 8, respectively) and the on-chip network during runtime, providing many levels for both. Howard et al. [1] provide some data on power consumption of Intel SCC, but do not focus on energy efficiency. Albers provides a survey on algorithmic-level techniques for energy-efficient computing [2]. Those techniques rely on a parameterized model of power consumption by cores. Thus, in order to be able to exploit the algorithmic techniques known in the art on the SCC, it seems helpful to derive the model's parameters.

In order to obtain those parameters, we devise microbenchmark programs and machine settings for the Intel SCC and measure the power consumption. We subject the differences between those measurements and the power model to a least-squares error analysis and thus derive the model parameters. In this way we provide a missing link between the abstract algorithmic-level treatment of power consumption and the hardware-oriented view of power consumption for SCC.

The remainder of this article is structured as follows. In Section II, we briefly review the model of power consumption used on an algorithmic level. In Section III, we present the experimental setting, the measurements and their analysis. Section IV discusses related work. Section V provides a conclusion and outlook on further work.

II. POWER CONSUMPTION MODEL

Viewed from an abstract level, the dynamic power consumption of a semiconductor device (such as a processor core) is dependent both on the frequency with which the device is clocked, and on the supply voltage. Thus, for a specific device we might model its dynamic power consumption at frequency f (assuming a fixed supply voltage) by

$$p_{dyn}(f) = b \cdot f^a \ , \tag{1}$$

where b and a are device-specific constants. Typically, a is assumed to lie between 2 and 3 [2]. A semiconductor device normally is also assumed to have a frequency-independent static power consumption $p_{stat} = s$, where s is a device-specific constant. The total power consumption of the device at a fixed supply voltage then sums up to

$$p(f) = p_{dyn}(f) + p_{stat} = b \cdot f^a + s \ .$$

The static power consumption is ignored in most studies as it used to be only a minor fraction of the total power consumption. However, its importance is growing due to shrinking feature size, even in embedded systems [3].

A similar formula can be derived for the dependency on the supply voltage, given a fixed frequency. However, the two parameters voltage and frequency are not independent, as the minimum and maximum possible frequencies depend (among other things) on the supply voltage. To simplify our investigation, we will concentrate on the clock frequency in accordance with [2]. To still incorporate the voltage, we assume that for each frequency, the least possible supply voltage is used. This leads to more efficient energy use and to a more accurate model while still sticking to only one operating parameter. As the static power consumption is linear in the supply voltage, given that one does not approach the threshold voltage too much [4, Eq. 10] and as the minimum possible supply voltage for a given frequency on SCC can be approximated by a linear relationship, we get

$$p(f) = p_{dyn}(f) + p_{stat} = b \cdot f^a + s \cdot f \ . \tag{2}$$

In order to compute the energy consumption during a time interval $[t_1; t_2]$, we can either multiply the power consumption with the length $t_2 - t_1$ of the time interval if the power consumption remains constant during that interval, or we have

to split the interval into sub-intervals with constant power consumption and sum up the energy consumptions of the sub-intervals otherwise. As the power consumption turned out to be approximately constant during our experiments of fixed length, energy and power consumptions are proportional, and thus only power is considered further.

The algorithmic techniques reviewed by Albers [2] either assume that frequencies can be scaled continuously, or that a finite number of discrete frequency levels are available. Also, some techniques use the features that cores can be put into energy-saving mode or even switched off if not needed. While frequency change is normally considered to happen atomically, i.e. without a time or power penalty, switch-off of cores implies both. Yet, we will restrict ourselves to frequency scaling and not consider switch-off in our current investigation.

The Intel SCC consists of 24 dual-core tiles, i.e. 48 cores. The cores are organized in 24 frequency islands, one for each tile with 2 cores, and 6 voltage islands each comprising 8 cores, for the purpose of dynamic frequency and voltage scaling. In order to change the frequency and voltage during the runtime of a program, the RCCE library, which is provided with the SCC, offers some power management functions. It is possible to change the frequency and voltage of the different islands separately. Another and, for our experiments, more important possibility is to change only the frequency of the cores and let the voltage automatically scale to the lowest stable state. In this case there are only 6 so-called *power domains*, which are equal to the voltage islands [5]. For changing the frequency, one has to set a *frequency divider* between 2 and 16 for each power domain. The frequency divider is an integer value with which the global reference clock frequency of 1.6 GHz is divided.

Thus we can vary the frequency (and voltage) as depicted in Tab. I.

TABLE I
TILE FREQUENCIES, RCCE FREQUENCY DIVIDERS AND VOLTAGES [6, PP. 39-40]

Tile Frequency (MHz)	RCCE Frequency Divider	Voltage
800	2	1.1
533	3	0.8
400	4	0.7
320	5	0.7
266	6	0.7
228	7	0.7
200	8	0.7
178	9	0.7
160	10	0.7
145	11	0.7
133	12	0.7
123	13	0.7
114	14	0.7
106	15	0.7
100	16	0.7

The cores are interconnected by an on-chip network, which also connects them to 4 on-chip memory controllers. Both on-chip network and memory controllers are frequency islands of their own. At system start, the on-chip network can be scaled at frequencies 1.6 GHz and 800 MHz [6, p. 22]. The memory controllers' frequency normally is not scaled because of their connection to the off-chip memory banks that need a fixed operating frequency. Thus in the following, we assume the memory controllers' power consumption to be static. This is a simplifying assumption, as the power consumption not only depends on voltage and frequency, but also on the application. If the cores and network are scaled down, while the memory controllers keep their frequency, then from the controllers' perspective the application changes, i.e. does fewer accesses [1].

In the following, we will denote by indices c, n and m functions or constants belonging to cores, network, and memory controllers, respectively. Thus, if the 6 power domains are run at frequencies f_1 to f_6 and the on-chip network at frequency f_0, then the power consumption of the Intel SCC would be modelled as

$$\begin{aligned} p_{scc}(f_0,\ldots,f_6) &= p_n(f_0) + p_m + \sum_{i=1}^{6} 8 \cdot p_c(f_i) \\ &= b_n \cdot f_0^{a_n} + 8 \cdot \sum_{i=1}^{6} b_c \cdot f_i^{a_c} \\ &\quad + s_n \cdot f_0 + s_m + 8 \cdot \sum_{i=1}^{6} s_c \cdot f_i \quad . \end{aligned} \tag{3}$$

The experiments of the following section are designed to experimentally derive the numerical values of b_n, b_c, a_n, a_c, s_n, s_m, and s_c. Normally, algorithmic power models [2] only consider the power consumption of complete chips. Therefore, we reduce the number of parameters by assuming $a_n = a_c$.

III. EXPERIMENTAL ANALYSIS

In each experiment, we fix the frequencies $f_0, \ldots, f_6$, run a microbenchmark program with one of four different settings on the SCC, and measure the power consumption one thousand times during the run of the program. For our measurements we use the FPGA on the Rocky Lake Board, which supports the direct measurement of voltages and currents of the domains. The program can read the actual voltage/current values through FPGA register access. Also, each experiment is repeated five times. We then compute the average power consumption from all measurements in each repeated experiment. When we put the frequency values and the measured power value into Eq. (3), we get a non-linear equation with the six unknowns b_n, b_c, a_n, s_n, s_m, s_c.

We devise a large number of frequency settings as explained below, and thus get a corresponding number of equations. We compute approximate values for the unknowns with a least-squares error analysis. Put shortly, for a set of values c_n to s_c and a set of values f_0 to f_6, the power consumption p_{scc} computed by Eq. (3) differs from the power consumption p_{exp} measured in the experiment, and thus produces a squared error $(p_{scc} - p_{exp})^2$. The analysis derives a set of values c_n to s_c such that the sum of the squared errors from all experiments is minimized.

Note that in our preliminary experiments, we did not vary the network frequency f_0, and thus the term $p_n(f_0)$ can also be considered static, which reduces the number of unkowns, and results in the following equation.

$$p_{scc}(f_1, \ldots, f_6) = 8 \cdot \sum_{i=1}^{6} (b_c \cdot f_i^{a_c} + s_c \cdot f_i) + \tilde{s} \ , \quad (4)$$

where $\tilde{s} = p_n(f_0) + s_m$.

We will explain below how to split $\tilde{s}$ into its components with the help of power measurements from [1]. In order to compute the parameters as accurately as possible, we decided to use a large range of frequencies. In order to restrict the number of experiments, we split the cores into two groups of sizes $8k$ and $48-8k$ each ($k = 0, \ldots, 6$), where one group has a higher frequency and the other group has a lower frequency. The cases $k = 0$ and $k = 6$ are special cases in which all cores have the same (higher or lower) frequency. We choose the frequencies 800 MHz, 533 MHz and 400 MHz for the high-frequency groups and 200 MHz and 100 MHz for the low-frequency groups. This results in $5 \cdot 3 \cdot 2 + 5 = 35$ experiments.

As a microbenchmark program we implemented a RCCE program whose execution can be divided into two steps that are detailed further below. In this program, one of the 48 cores is the *coordinator*, which coordinates the groups and measures the power consumption, while the other ones are the slaves. It is important to note that in every power domain only one core, the *power domain master*, can change the frequency and voltage of the corresponding domain. Hence, there are 6 power domain masters and one coordinator, which is one of the power domain masters.

The user starts the microbenchmark program with four parameters: the number of high-frequency domains, the high-frequency divider, the low-frequency divider and the setting for the microbenchmark. The settings allow to have several microbenchmarks, that differ in the use of caches, and intensity and regularity of memory access. There are four settings in total (cf. table II). In the first setting only one integer variable is used, which is initially set to 0. In the other settings we use an array with one million elements. These elements are initially set to 0 for the second setting, to the maximum integer value for the third setting and to the index of the element in the fourth setting.

TABLE II
DIFFERENT BENCHMARK SETTINGS

Benchmark	Description
0	Step 1: One variable, initially set to 0 Step 2: Variable is incremented by 1
1	Step 1: array[size 10^6], initially set to 0 Step 2: array elements added up successivly
2	Step 1: array[size 10^6], initially set to max_int Step 2: array elements added up successivly
3	Step 1: array[size 10^6], initially set to index Step 2: array elements added up in the following order: (7· index + rank) mod array_size

In the first step of the program all cores verify if they are a power domain master or not, and send the result (my_rank if they are a power domain master, -1 otherwise) to the coordinator. The coordinator saves the ranks of the power domain masters in an array and sends to each power domain master its array index, incremented by 1. In this way, we make sure that each power domain master has a unique *domain-master rank* between 1 and 6. Thus each power domain master can, from the user-defined number of high frequency domains, find out to which group its power domain belongs to. After that, the power domain masters of both frequency groups scale the frequency (and thus also the voltage) to the user-defined frequency. At the end of step 1, all cores are synchronized by a barrier to make sure that all frequency and voltage scalings for both groups have finished and all cores can begin with the second step simultaneously.

In the second step we use a time controlled loop and simulate an expensive calculation on each core for 10 seconds. The calculation depends on the chosen microbenchmark setting. In the first setting the integer variable is incremented by 1 within the loop. This represents a microbenchmark with use of ALU and caches, and few memory accesses. The second and third settings add up the array elements successively. This represents microbenchmarks with cache and memory accesses, and different ALU use (adding up zeroes, adding up ones). The last setting uses a more unstructured access pattern (cf. table II) to add up the array elements, which represents higher cache miss rate and thus higher memory traffic. While working in the loop, the coordinator also measures the power consumption of the whole SCC every 10th millisecond. Thus we get an averaged result over 1000 measurement points for each microbenchmark setting. In this way we obtain many results for different situations and can compare these results to each other.

Figure 1 shows the results of the four microbenchmark settings for a high frequency of 800 MHz and the low frequencies of 100 MHz and 200 MHz, respectively. We can see that the results of the different microbenchmark settings are very similar to each other. The results for the other core frequencies (533 MHz and 400 MHz) differ only in that with decreasing the high frequency the power consumption also decreases with an increasing number of high frequency domains. Thus we only show the results for 800 MHz as an example for all results that we obtained. Also the workload over the 1000 measurement points within each experiment is quite stable. It mostly varies in a range of 2 Watt. There are less than 5 outliers on each experiment.

Figure 1 depicts the average power consumption from each of the 35 experiments. By a least-squares error analysis of equations obtained by inserting the frequency values and a fixed value of 3 for a_c into Eq. (4), we obtain the values $b_c \approx 2.015 \cdot 10^{-9}$ Watt/MHz^{a_c}, $s_c \approx 10^{-6}$Watt/MHz and $\tilde{s} \approx$ 23 Watt. The average error is 1.95 Watt, which is 5.58% of the average power consumption (averaged over all experiments). The relative error ranges from -14.66% to 24.73%. Thus, our model matches the experimental data quite well.

The comparison between measured and modelled power

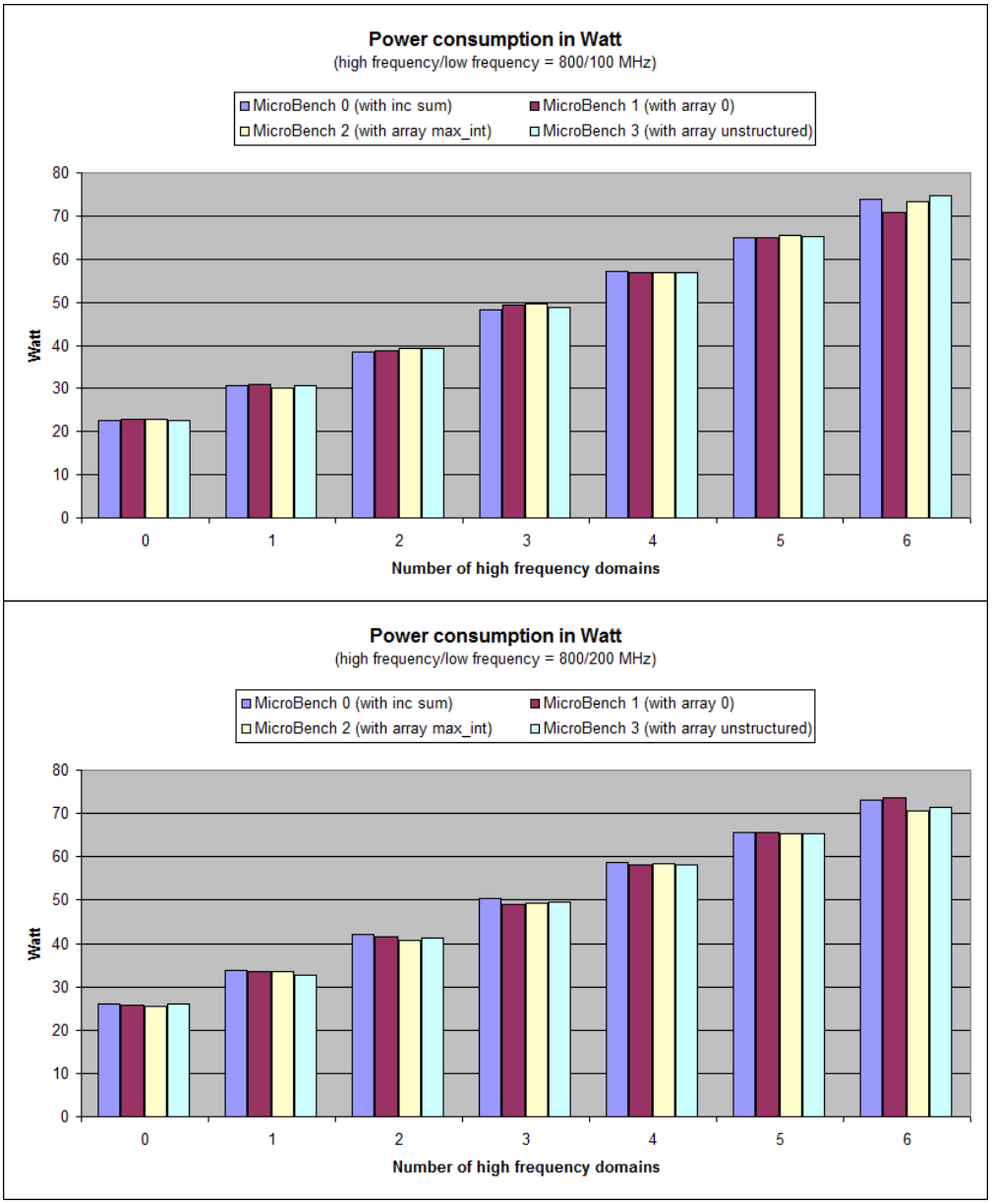


Fig. 1. Power consumption of the SCC for all microbenchmark settings with a different number of high-frequency domains, for the frequency 800 MHz. The low frequency is kept at 100 MHz (top) and 200 MHz (bottom), respectively.

consumption is depicted in Fig. 2. In this figure each curve or peak represents the results of one microbenchmark for the different numbers of high frequency domains. There are totally 24 peaks, which are organized clockwise as follows: On the right side of the circle each pair of four peaks are the results of the four microbenchmarks with a high frequency of 800 MHz, 533 MHz and 400 MHz (from top to bottom) and with a low frequency of 100 MHz. On the left side of the circle the peaks are represented in a reverse order (from bottom to top) and with a low frequency of 200 MHz.

Note that the measurements for all cores running at the same frequency ($k = 0$ or 6) correspond quite well to the numbers reported in [1, Fig. 13]. There cannot be an exact comparison as we do not know their benchmark program.

According to [1, Fig. 14], the network and memory controllers consume between 18.4 Watt (at cores 125 MHz, network 250 MHz) and 35.7 Watt (at cores 1 GHz, network 2 GHz). As we do not change the frequencies for network and memory controllers in our experiments, the value for the static power consumption of around 23 Watt seems to match an interpolation between those values quite well. One has to take into account that the frequency for the network and memory controllers in our experiments is fixed to only 800 MHz for both. Thus the voltage of these components is around 0.75 V

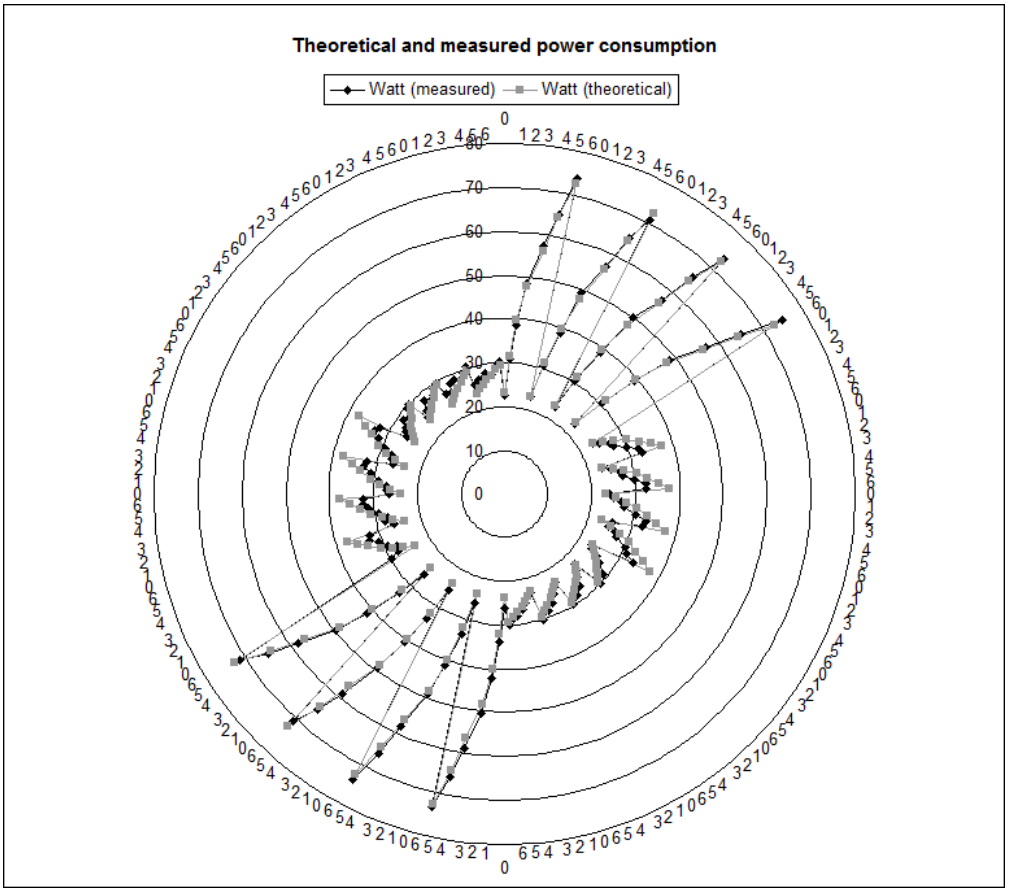


Fig. 2. Comparison of measured and modelled power consumption after least-squares error analysis.

and the static power consumption is as expected nearer to the lower bound of 18.4 Watt. As we can only measure the power consumption of the complete chip, we cannot avoid to incorporate measurements done by the chip manufacturer [1].

IV. Related Work

Ioannou et al. [7] consider dynamic voltage and frequency scaling for the SCC and implement a hierarchical power management with phase prediction to minimize energy consumption, balancing the trade-off between energy consumption and execution time for the various computation phases of MPI programs at runtime. Their approach is not based on using an explicit power model derived off-line, but on iterative adaptation of frequency and voltage within a performance window to react to changed computation patterns. The dynamic adaptation takes into account information for the local power domain as well as for groups of multiple power domains. They report on average improvements of 11.4% in the energy-delay product, with an average increase in execution time by 7.7% compared to running constantly at maximum frequency.

Putigny et al. [8] propose a performance and power model for SCC based on the core frequency. The model can be used for predicting the behavior of regular code such as dense linear algebra kernels that can be suitably characterized by a few statically accessible parameters. Their power model does not consider static power and does not model the network power. The constant coefficient of the dynamic energy consumption is not derived, and the scenario used here of combining voltage and frequency is not mentioned.

Gschwandtner et al. [9] investigate the impact of core frequency and voltage on the performance and power consumption of three major benchmark applications on SCC. They find that the benefits of core frequency and voltage scaling depends on the program, in particular whether it is compute-bound. They also demonstrate that, depending on the application, the energy-optimal frequency for an entire application can be at an intermediate level, both lower than the maximum possible frequency (which yields minimum execution time) and higher than the minimum possible frequency (which yields minimum power usage). In contrast to our approach, their work is based on measurements for entire applications and does not derive an energy model that could be used for predictions to support optimizations at a fine-grained level.

Kiertscher et al. [10] present an energy saving daemon for clusters called cherub, which can interact with different resource management systems to make them energy-aware. In this daemon they use the most important modes from standards and specifications like the Intelligent Platform Management

Interface (IMPI) or the Advanced Configuration and Power Interface (ACPI) specification to save energy. The cherub daemon polls the state of the cluster in regular intervals to gather the state of the different nodes and the load situation. With this imformation it can execute some actions to switch the states of the nodes and also distribute their load to other nodes. In this way cherub makes the use of a cluster more efficient in terms of workload and energy.

V. CONCLUSIONS AND FUTURE WORK

Our research provides the SCC-specific parameters for power consumption models used in algorithmic research on energy efficiency, thus providing a missing link between both worlds.

For the future, we plan to include changes of the network frequency to get more accurate measurements and analyses, and to use the insights gained in the present experiments to derive a power-optimal algorithmic mapping of streaming applications onto the Intel SCC.

We also plan to extend the model to include situations where cores can be switched off completely, which may be possible in future architectures [2].

Finally, we would like to refine our timing measurements to derive time penalties for changing voltage or frequency to widen the algorithmic applicability of our results.

Our power model considers the SCC at a quite high level only, and makes some simplifying assumptions to reduce the model complexity. A more detailed model constructed from microarchitectural simulation or analysis would be beyond the scope of our work that focuses on algorithmic level power modeling. Instead, a complementary, microarchitecture-agnostic approach could start from a generic but detailed algorithmic power model that includes many possibly relevant terms, involving parameters both from the algorithmic and architectural level, and which may or may not be so obvious from a high-level view onto the chip itself and its environment. The coefficients of this model could be calibrated by regression analysis (or other machine learning techniques) over training runs of benchmark programs with different characteristics, such as the Berkeley Dwarfs [11, Sect. 3] or similar computational kernels. This process automatically creates a SCC-specific power model from training data that can be used for algorithmic-level power predictions instead of our simplified model, for instance when deciding between different applicable algorithmic variants, resource allocation, choice of tunable parameters, or optimizing transformations for a computation. Similar techniques have been used for automatic model construction in other contexts, e.g. by Steinke et al. [12] for instruction-level energy modeling of an ARM7 processor or by Brewer [13] for algorithmic-level execution time modeling.

Another way of algorithmic-level power modeling can be applied after refactoring applications as combinations (such as serial, parallel and hierarchical composition) of a small set of algorithmic building blocks representing characteristic computational *patterns* [14] (where generic patterns are also known as *skeletons* in the literature [15]). For each such skeleton, a power model might be defined by interpolation of measurement tables or auto-calibration of a given generic model, and hence the power model for the overall application would be composed accordingly from the skeletons' power models

ACKNOWLEDGMENTS

The authors thank Intel for providing the opportunity to experiment with the "concept-vehicle" many-core processor "Single-Chip Cloud Computer". C. Kessler acknowledges partial funding by Vetenskapsrådet for this work.

REFERENCES

[1] J. Howard, S. Dighe, S. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, and R. Van Der Wijngaart, "A 48-Core IA-32 message-passing processor in 45nm CMOS using on-die message passing and DVFS for performance and power scaling," *IEEE J. of Solid-State Circuits*, vol. 46, no. 1, pp. 173–183, Jan. 2011.

[2] S. Albers, "Energy-efficient algorithms," *Comm. ACM*, vol. 53, no. 5, pp. 86–96, May 2010.

[3] A. Andrei, "Energy efficient and predictable design of real-time embedded systems," Ph.D. dissertation, Linköping University, The Institute of Technology, 2007, linköping Studies in Science and Technology, No. 1127.

[4] A. P. Chandrasakaran and R. W. Brodersen, "Minimizing power consumption in digital CMOS circuits," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 498–523, Apr. 1995.

[5] Intel Labs, "Using the RCCE power management calls," Sep. 2011.

[6] ——, "The SCC programmer's guide," Nov. 2011.

[7] N. Ioannou, M. Kauschke, M. Gries, and M. Cintra, "Phase-based application-driven power management on the single-chip cloud computer," in *International Conference on Parallel Architectures and Compiler Techniques (PACT 2011), Galveston, Texas, USA*. ACM, 2011.

[8] B. Putigny, B. Goglin, and D. Barthou, "Performance modeling for power consumption reduction on SCC," in *Proc. 3rd Many-core Applications Research Community Symposium (MARC-3)*, Dec. 2011.

[9] P. Gschwandtner, T. Fahringer, and R. Prodan, "Performance analysis and benchmarking of the Intel SCC," in *Proc. 2011 IEEE Int. Conf. on Cluster Computing*. IEEE Computer Society, Sep. 2011, pp. 139–149.

[10] S. Kiertscher, J. Zinke, S. Gasterstädt, and B. Schnor, "Cherub: Power consumption aware cluster resource management," *IEEE/ACM GreenCom-CPSCom*, pp. 325–331, 2010.

[11] K. Asanovic *et al.*, "The landscape of parallel computing research: A view from Berkeley," Electrical Engineering and Computer Sciences, University of California at Berkeley, USA, Tech. Rep. UCB/EECS-2006-183, Dec. 2006.

[12] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel, "An accurate and fine grain instruction-level energy model supporting software optimizations," in *Int. Workshop on Power and Timing Modeling, Optimization and Simu lation (PATMOS)*, Sep. 2001.

[13] E. A. Brewer, "High-level optimization via automated statistical modeling," in *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP'95)*, 1995.

[14] C. W. Keßler, "Pattern-driven Automatic Parallelization," *Scientific Programming*, vol. 5, pp. 251–274, 1996.

[15] M. I. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman and MIT Press, 1989.

 ISBN: 978-2-7257-0016-8

Transparent Programming of Many/Multi Cores with OpenComRTOS

Comparing Intel 48-core SCC and TI 8-core TMS320C6678

Bernhard H.C. Sputh, Andrew Lukin and Eric Verhulst
Altreonic NV
Gemeentestraat 61A Bus 1; B3210 Linden; Belgium
Email: {bernhard.sputh, andrew.lukin, eric.verhulst}@altreonic.com

***Abstract*—Developing software for non SMP multi-core systems such as the 48 core Intel-SCC or the TI-TMS320C6678 is a complex task, and will become even harder with the emerging heterogeneous multi-core systems combining different architectures on a single chip. To tackle this issue, Altreonic has adopted a formalized approach to embedded systems development. Of particular interest is the formally developed OpenComRTOS, that allows one to program distributed systems ranging from single node microcontrollers, over multi-core to networks of heterogeneous networked processing nodes, in a fully transparent way. The current implementation can theoretically handle 2^{24} nodes. Together with its tools it provides the core of OpenComRTOS Designer.**

This paper reports the results of porting OpenComRTOS to the Intel-SCC, i.e. code size and performance figures comparing them with other ports, with a focus on the TI-TMS320C6678. Furthermore, it describes the basic structure of the OpenComRTOS Intel-SCC port, focussing on the inter-core communication.

I. INTRODUCTION

Users of embedded systems continuously expect more features. At the same time processors are becoming cheaper and more powerful. However, user expectations rise faster than the progress of the hardware. Hence, the evolution to multi/many-core architectures while being enabled by technology advances, is also a solution to achieve more performance at less energy costs. The question is, how to program them?

OpenComRTOS [1], [2] was designed from the start to address this issue. Building on the concepts of CSP [3], Hoare's Process Algebra and the experience with a previously developed parallel RTOS (Virtuoso) [4], formal modelling was used. The top level requirements were to achieve a transparent concurrent programming model for real-time embedded systems. This was called the "Virtual Single Processor" programming model. At the API level, a program is composed of "tasks", each having a private workspace and priority. Task synchronise and communicate using instances of "Hubs". As such, Hubs are instantiated to the traditional RTOS services like Events, Semaphores, FIFO, Resources, etc.

OpenComRTOS is build as a scheduler on top of a prioritised packet switching and communication layer. It is designed to run on heterogeneous systems thus a heterogeneous set of nodes, connected using a heterogeneous set of communication means (shared memory, fast point-to-point links, or switching networks). To support this the programming approach separates the network topology from the application topology, allowing cross development or simulation on single node systems (like a PC). Once a program has been developed its entities (Tasks and Hubs) can be remapped to a different topology without source code changes. Only a recompilation is needed and maybe some I/O drivers will need to be modified. This is achievable because the hubs, used by tasks to interact, are decoupled from the tasks.

The Intel-SCC [5] is an experimental system which consists of 48 Pentium cores which are inter-connected over a routing network. This routing network also connects the cores to the four on-chip memory controllers, which support up to 64GB of memory in total. Texas Instruments provides the TMS320C6678 [6], which is a commercially available 8 core DSP where the cores and the peripherals are interconnected using a bus called TeraNet. In the following we will refer to this chip as TI-C6678. In this paper we compare the performance figures of OpenComRTOS on both architectures.

The rest of the paper is organised as follows, Section II details the architecture of OpenComRTOS, and how applications are developed for it. This is followed by the implementation details of the OpenComRTOS port to the Intel SCC in Section III. Section IV compares the Intel-SCC port with other ports of OpenComRTOS. The paper closes with Conclusions and Further Work in Section V.

II. OPENCOMRTOS PARADIGMS

OpenComRTOS uses the two following paradigms: "Interacting Entities", discussed in Section II-A and "Virtual Single Processor" which is discussed in Section II-B.

A. Interacting Entities

OpenComRTOS has a semantically layered architecture. Table I provides an overview of the available services at the different levels. At the lowest level the minimum set of Entities provides everything that is needed to build a small networked real-time application.

There are two types of Entities in OpenComRTOS: active and passive Entities. Active Entities are Tasks (having a private function and workspace), passive Entities are Hubs, used to synchronise and communicate between Tasks (see Figure 1).

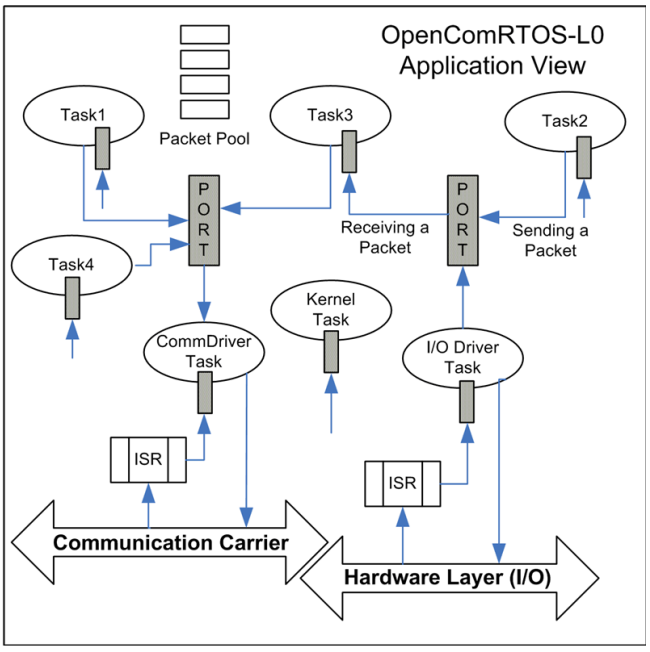


Fig. 1. OpenComRTOS-L0 view

While there are different types of Hubs in OpenComRTOS, the most fundamental one is the *Port*. The Port-Hub acts like a channel in the sense of Hoare's CSP, but allows multiple waiters and asynchronous communication.

One of the Tasks is a Kernel Task which schedules the other Tasks in order of priority, manages Hub-based services, and routes Packets. Driver Tasks handle inter-node communication. Pre-allocated as well as dynamically allocated Packets are used as carriers for all activities in the RTOS, such as: service requests to the kernel, Hub synchronisation, data-communication, etc. Each Packet has a fixed size header and a data payload with a user defined but global data size. This significantly simplifies the Packet management, particularly at the communication layer. A router function transparently forwards Packets in order of priority between the network nodes. The priority of a Packet is the same as the priority of the Task from which the Packet originates, except when the priority inheritance algorithm changes the priority of the task after the package was sent.

In the next semantic level Services and Entities were added, similar to those found in most RTOSs: Boolean events, counting semaphores, FIFO queues, resources, memory pools, etc. The formal modelling lead to the definition of all these Entities as semantic variants of the common and generic "Hub" entity. In addition, the formal modelling also helped to define "clean" semantics for such services, whereas ad-hoc implementations often have side-effects. Table II summarises the semantics.

The services are offered in a non-blocking variant (_NW), a blocking variant (_W), and a blocking with time out variant (_WT). All services are topology transparent and there is no restriction in the mapping of Task and kernel Entities onto the network. See Tables II and III for details on the semantics.

Using a single generic Hub entity leads to more code reuse, therefore the resulting code size is at least 10 times less than for an RTOS with a more traditional architecture. One could of course remove all such application-oriented services and just use Hub based services. Unfortunately, this has the drawback that services loose their specific semantic richness, e.g. resource locking clearly expresses that the Task enters a critical section in competition with other Tasks. Also erroneous run-time conditions are easier to identify at application level.

During the formal modelling process, we also discovered weaknesses in the traditional way priority inheritance is implemented in most RTOSs. Fortunately, we found a way to reduce the total blocking time. In single processor RTOS systems this is less of an issue, but in multi-processor systems, all nodes can originate service requests and resource locking is a distributed service. Hence, the waiting lists can grow longer and lower priority Tasks can block higher priority ones while waiting for the resource. This was solved by postponing the resource assignment until the rescheduling moment. Finally, by generalisation, also memory allocation has been approached like a resource locking service. In combination with the Packet Pool, this opens new possibilities for safe and secure memory management, e.g. the OpenComRTOS architecture is free from buffer overflow by design.

For the third semantic layer (L2), we plan to add dynamic support like mobility of code and of kernel Entities. A potential candidate is a light-weight virtual machine supporting capabilities as modelled in pi-calculus [7]. For this purpose we developed the Safe Virtual Machine [8], which currently allows to dynamically load and execute tasks.

B. *Virtual Single Processor Programming Model*

The Virtual Single Processor (VSP) programming model of OpenComRTOS, provides the user with the ability to treat a complex network of processors like a single one. An OpenComRTOS project consists of the following elements, defined in a graphical modelling environment.

1) Topology — A topology in OpenComRTOS defines the hardware of the system: processing Nodes, the communication Links between them and peripherals. Each Node has an unique name. Furthermore, in the topology the Node specific settings are defined, such as the compiler to use, and the configuration of device drivers. At any time of the development process it is possible to modify the number of Nodes in the Topology. Thus the user can start using just one Node, and later on gradually increase the number of Nodes.
2) Application — Here the user specifies which Entities are used in the project and what interactions they perform. Each Entity (Tasks and Hubs) has an unique name in the system and is mapped onto one Node of the topology. This mapping can be changed at any time during the development process, as long as no Node-dependencies were inserted. A typical node dependency is a Task that directly accesses peripherals of a specific Node. The

TABLE I
OVERVIEW OF THE AVAILABLE ENTITIES ON THE DIFFERENT LAYERS

Layer	Available Entities
L0	Task, Hub (instantiated as Port)
L1	Task, Hub instantiated as: Port, Boolean Event, Counting Semaphore, FIFO Queue, Resource, Memory Pool or user defined
L2	Mobile Entities: all L1 entities are moveable between Nodes.

TABLE II
SEMANTICS OF L1 ENTITIES

L1 Entity	Semantics
Event	Synchronisation on a Boolean value.
Semaphore	Synchronisation with counter allowing asynchronous signalling.
Port	Synchronisation with exchange of a Packet.
FIFO queue	Buffered, asynchronous communication of Packets. Synchronisation on queue full or empty.
Resource	Event used to create a logical critical section. Resources have an owner Task when locked.
Memory Pool	Linked list of memory blocks protected with a resource.

TABLE III
SERVICE SYNCHRONIZATION VARIANTS

Services variants	Synchronising Behaviour
_NW	Non Waiting: when the synchronisation fails the Task returns with a RC_Failed.
_W	Waiting: when the synchronisation fails the Task waits until such event happens.
_WT	Waiting with a time-out. Waiting is limited in time defined by the time-out value.

logical behaviour of the system is independent of the mapping of the Entities, only the latency may change.

The unique name of an Entity is used for addressing the Entity in the system. Internally an Entity-ID gets used to interact with an Entity. This Entity-ID consists of two components: the global Node-ID, and a local ID, both ID's get generated at build time, by the code generators. When remapping an Entity, to a different Node, the Entity-ID will change, while the unique name stays the same. This addressing scheme and the use of Packets to represent Task interactions (service calls), results in a programming model where the Entities can be placed anywhere. The OpenComRTOS kernel Task acts as a switch, sending the Packets to their destination. Note, that the same mechanism is used for local as well as remote Entities. It is this packet switching nature of OpenComRTOS that makes it so scalable. All code is multi-processor by default. Note also that the user is largely relieved from the tedious effort of writing all data structures and initialisation routines. This is largely generated from higher level descriptions and topology metamodels in the graphical OpenComRTOS Designer modelling environment.

III. PORTING OPENCOMRTOS TO INTEL SCC

Due to its architecture, with a a clean separation between the HAL (Hardware Abstraction Layer) and the operating system services OpenComRTOS is fairly easy to port to a new platform. We have done basic ports (single Node with periodic timer) to new architectures, such as to the NXP-CoolFlux [9], and the TI TMS320C6678 [6] within two weeks. The most difficult part of the porting effort is usually to integrate the toolchain to compile the code with it. This assumes of course that adequate documentation and tools are available.

In case of the Intel SCC it took longer than the usual two weeks due to the experimental nature of the development support. Nevertheless, even while only having remote access to the hardware, once the chip's hardware was understood and the basic functionality was implemented, development was straightforward.

A. SCC-Bringup

We use the Bare Michael framework [10] as underlying library to bring up the individual cores of the Intel-SCC. Once execution reaches the `main()` function OpenComRTOS initialises the Tasks, and starts the communication drivers, before starting the Kernel-Task.

B. Inter Core Communication

OpenComRTOS is designed to allow the development of distributed heterogeneous systems. This means that it provides the capability to build systems consisting of multiple CPUs interconnected over various communication means, such as RS232, Ethernet, shared memory and now also the Intel-SCC Message Passing Buffers (MPB). The communication between different Nodes of the system is handled by so-called transfer-packets, which have system wide the same structure. The transfer-packet consist of a 32 B header and a variable amount of payload data. When a Task issues a service request to a Hub that is located on another Node, then the kernel-task routes the service request packet to the corresponding Link Driver to transfer it to its destination Node. The routes are precalculated during the build process and do not change during run-time, relieving the application from any explicit routing.

In the Intel-SCC each tile, which consists of two cores, provides a 16 kB large Message Passing Buffer (MPB). In the link driver implementation we assigned each core of the tile

8kB of this buffer which it uses as an input port for the link driver. This means that each core reads the messages meant for it from its part of the MPB. To send a message each core writes the message directly into the MPB of the core the message is intended for, i.e. we establish a full mesh on the Intel-SCC, leaving all the routing decisions to the underlying routing network. Inside the MPB the data is organised using a lock free ring buffer implementation, where the writer and reader task do not need to lock each other out. However, it is still necessary to prevent that more than one writer tries to gain access to the MPB in parallel, thus there is one locking operation involved. The lock is represented by an atomic variable, and we use the `acquire_lock()` and `release_lock()` functions, provided by Intel, to manipulate it. Having an RTOS means that it is necessary to inform the reader core that new data has arrived, this is achieved by the writer-node issuing an Interrupt Request (IRQ) to the reader-node, using the function `interrupt_core()`. Upon receiving the IRQ, the reader-node reads out the data, translates the transfer packet into a local packet and and then passes it to the kernel-task for processing.

IV. MEASUREMENT RESULTS

OpenComRTOS has been ported to quite a number of different CPU architectures already. In this section we compare codesize and performance figures of the Intel-SCC port with the figures of selected other ports. All measurements with the Intel-SCC system were done using the following configuration: core: 533 MHz, memory: 800 MHz, mesh: 800 MHz. To allow the cache to initialise on the Intel-SCC the first measurement in each of the following benchmarks was ignored.

A. Code Size

Table IV gives detailed code size figures, in byte, for our currently available ports of OpenComRTOS. The Intel-SCC port has a typical code size for a 32 bit instruction set machine, similar to the MicroBlaze, Leon3, XMOS, and TI-C6678 ports we have done in the past.

B. Performance Figures

Next we consider the runtime performance. Table V states the elapsed time to perform what we call a semaphore loop (two task signalling each other in a loop using two semaphore hubs, [1] gives an explanation, Figure 2 shows the application diagram). This test gives a very good indication of the latencies introduced by the OS and gives a good indication of task scheduling and service request latencies as each loop consists of 4 context switches, and 4 service requests with a total of 8 Packet exchanges. The measurements were performed by measuring the loop time 1000 times, using the highest precision timer available in the system, in case of the NXP CoolFlux the cycle counter of the simulator was used. In all cases we tried to achieve top performance, thus available caches were utilised. Furthermore, interrupts were disabled, except the one for the periodic timer tick. The column 'Context Size' of the table gives the number of registers that has to be saved and the size of these registers, for a user triggered context switch. The context saved when handling an interrupt has a different size.

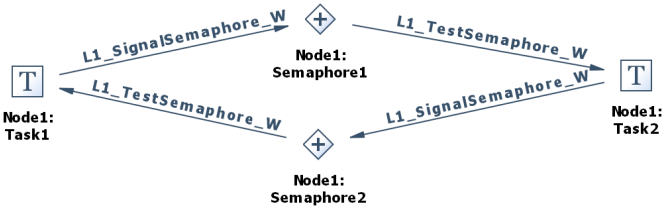


Fig. 2. Application Diagram of the Semaphore Loop Benchmark

C. Interrupt Latency Measurements

Another important performance figure, for an RTOS, is the interrupt latency. We differentiate two types of latencies: IRQ (Interrupt ReQuest) to ISR (Interrupt Service Routine), and IRQ to Task. The first one measures how long it takes after an automatic reload counter issued an IRQ until the first useful instruction can be performed in the ISR, this means that all context saving has been performed already. The IRQ to Task latency represents how long it takes until a high priority task can perform the first useful instruction after an IRQ has occurred. However, these are no single figures because it depends on what the CPU is currently doing. Thus we collected a few million measurements, and performed a statistical analysis of them. Table VI gives the minimal, maximal and the median (50% value of all measured latencies).

For the Intel-SCC and the TI-C6678 system we presently have only the minimal figures for an unloaded system. We have the following interrupt latencies for these platforms:

- Intel-SCC:
  - IRQ to ISR: 656.78 ns (349 cycles)
  - IRQ to Task: 10.32 µs (5501 cycles)
- TI-C6678:
  - IRQ to ISR: 136 ns (136 cycles)
  - IRQ to Task: 1.37 µs (1367 cycles)

Both the Intel-SCC and the TI-C6678 have a larger interrupt latency, in number of cycles, than e.g. the ARM-Cortex-M3, however they are clocked at a much higher clock speeds thus the absolute times are better. However, it is clear that the Intel-SCC was not designed for realtime applications, unlike a micro controller such as the ARM-Cortex-M3. The ARM-Cortex-M3 does a lot of the necessary task saving and restoring, as well as interrupt dispatching operations, using dedicated hardware, while in case of the Intel-SCC and the TI-C6678 it has all to be done in software.

A point regarding the TI-C6678: this processor has multiple cascaded interrupt controllers (for a potential total of about 1000 interrupt sources), which have been taken out of the equation as we just measured the latency of C66x core internal interrupt controller, which provides 16 interrupts, of which 12 can be freely used for external interrupts.

TABLE IV
OPENCOMRTOS L1 CODE SIZE FIGURES (IN BYTES) OBTAINED FOR OUR DIFFERENT PORTS

Service	MLX16	MicroBlaze	ESA-Leon3	ARM Cortex M3	XMOS	TI-C6678	Intel-SCC
L1 Hub shared	400	4756	4904	2192	4854	5104	4321
L1 Port	4	8	8	4	4	8	7
L1 Event	70	88	72	36	54	92	55
L1 Semaphore	54	92	96	40	64	84	64
L1 Resource	104	96	76	40	50	144	121
L1 FIFO	232	356	332	140	222	300	191
L1 PacketPool	NA	296	268	120	166	176	194
Total L1 Services	1048	5692	5756	2572	5414	5908	4953

TABLE V
OPENCOMRTOS LOOP TIMES OBTAINED FOR OUR DIFFERENT PORTS

	Clock Speed	Context Size	Memory Location	Loop Time	Cycles
ARM Cortex M3	50 MHz	16×32 bit	internal	52.5 µs	2625
NXP CoolFlux	NA	70×24 bit	internal	NA	3826
XMOS	100 MHz	14×32 bit	internal	26.8 µs	2680
Leon3	40 MHz	32×32 bit	external	136.1 µs	5444
MLX-16	6 MHz	4×16 bit	internal	100.8 µs	605
Microblaze	100 MHz	32×32 bit	internal	33.6 µs	3360
TI-C6678	1 GHz	15×32 bit	L2-SRAM	4.5 µs	4500
Intel SCC	533 MHz	11×32 bit	external	4.9 µs	2612

TABLE VI
OPENCOMRTOS INTERRUPT LATENCIES ON AN ARM-CORTEX-M3 @ 50MHZ

	IRQ to ISR	IRQ to Task
Minimal	300 ns (15 cycles)	12 µs (600 cycles)
Maximal	2140 ns (107 cycles)	25 µs (1250 cycles)
50%	400 ns (20 cycles)	17 µs (850 cycles)

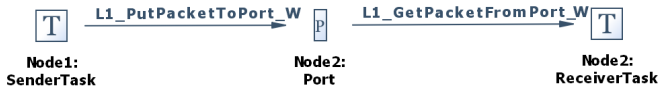


Fig. 3. Application Diagram for the Throughput Measurement

D. Inter Core Communication

To measure the application level inter core communication throughput, i.e. the usable task-to-task bandwidth when developing an application we performed the following measurements. The benchmark system consists of two tasks: a SenderTask and a ReceiverTask, communicating using a Port-Hub. Figure 3 shows the application diagram of the system. The SenderTask sends an L1-Packet to the Port-Hub from which the ReceiverTask receives it. The Port-Hub interactions are done using waiting semantics, which means that the SenderTask has to wait until the Receiver-Task has synchronised with it in the Port-Hub. The Port-Hub copies the payload data contained in the L1-Packet from the Sender-Task to the L1-Packet from the Receiver-Task, and then sends acknowledgement packets to both Tasks. We measured how long it takes the ReceiverTask to receive 1000 times a data packet of a specific size. To perform the initial synchronisation the ReceiverTask waits for a first communication to take place before determining the start time. Please note that the SenderTask and ReceiverTask synchronise in the Port-Hub, thus the SenderTask can only send the next packet, after it has received the acknowledgement packet that the previous transfer was performed successfully.

1) Intel-SCC: When distributing the Tasks over different Nodes in the system, the data will be transferred between the two nodes using link drivers and using the on-chip communication mechanism. These link drivers translate the L1-Packet to a Transfer-Packet, and transfer only the used part of the data part of the L1-Packet. We measured the following different system setups, with different payload sizes:

- Single-Core: In this setup all Tasks and the Hub are on the same core. Thus no inter core communication is involved.
- Multi-Core: Afterwards the benchmark was distributed over two nodes, in the following way:
  - Node1: SenderTask
  - Node2: ReceiverTask and Port-Hub

  In this setup we measured with different numbers of Hops (see [5] for details) between the two cores:
  - No-Hop: Node1 on core 10 and Node2 on core 11
  - 1-Hop: Node1 on core 10 and Node2 on core 8
  - 8-Hops: Node1 on core 10 and Node2 on core 36

Figure 4 gives the measured results for the different systems. What sticks out is that the single core example goes into saturation at around 20 MB/s, while the distributed versions achieve a higher throughput of up to 33 MB/s. These figures are similar to the ones reported by Lankes et. al. in [11]. There is also a strange jump in throughput from payload sizes 128 B to 256 B, for the distributed version, which we do not observe in the single core version. Furthermore, we see a strong influence of the routing network which nearly halves the throughput between the No-Hop and the 8-Hop versions, thus the location of the Nodes and their distance matters on the Intel-SCC.

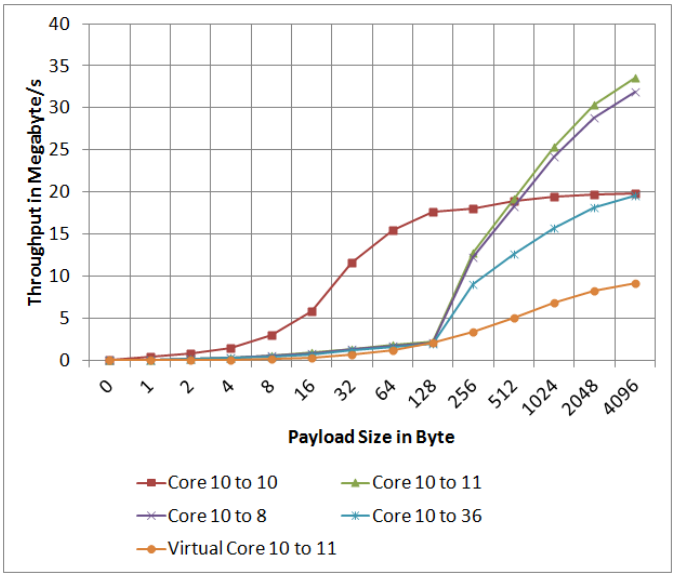


Fig. 4. Intel-SCC Throughput over Packet Payload Size

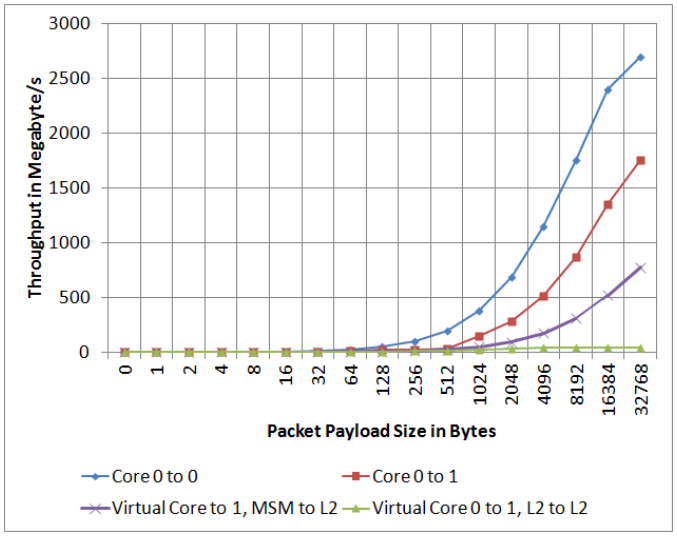


Fig. 5. TI-C6678 Throughput over Packet Payload Size

The curve labelled 'Virtual Core 10 to 11' is moving the data, by transferring the ownership of a shared buffer from core 10 to core 11. This is done by transferring the buffer information (address, size, resource-lock-id) from core 10 to core 11 using a port-hub. Once core 11 has this information it locks a resource, to avoid unintentional access, copies the data, and then releases the lock. The achieved throughput is about half of what we achieved in the single core version. The reason for this is that the buffer is placed in shared memory which halves the achievable throughput. The throughput of the bare version, i.e. without OpenComRTOS running, just a main and bare michael, drops from 17.4 MB/s, when copying from private memory to private memory, to 10.3 MB/s when copying from shared memory to private memory.

2) TI-C6678: The TI-C6678 evaluation board available to us was clocked at 1 GHz, thus all measurements were done at this frequency. Another point to mention is that none of the DMA units provided by the TI-C6678 have been used for these measurements, thus the DSP-Core had to spend all its cycles to move the data.

Figure 5 gives the throughput measurements for the TI-C6678 @ 1 GHz, for both the single core ('Core 0 to 0') and the distributed version ('Core 0 to 1'). A few words regarding the measurement setup. In case of the single core measurement, the data and the code where completely within the 512 kB large L2-SRAM of core 0. This is possible because the architecture permits to use the L2 cache as SRAM. For the distributed version we used the Queue Management Sub System (QMSS) queues [12] to transfer descriptors of transfer packets between the cores. The queues 652 and 653 were used, generating an interrupt when data is pending on them. The shared transfer packets were located in the Multicore Shared Memory (MSM), constituting 4 MB of fast memory shared between the cores. This memory is part of the Multicore Shared Memory Controller (MSMC) [13], which interfaces the eight cores to external DDR-SRAM. For the single core version we achieve a top throughput of 2695 MB/s using packets with 32 kB payload. The distributed version achieved a maximum throughput of 1752 MB/s with the same payload. In both cases we have not yet reached the saturation of the system, thus the total throughput will be higher, if we increase the packet payload size.

Like for the Intel-SCC we've also implemented a measurement of the virtual bandwidth, using a shared buffer. With a buffer size of 32 kB we achieved a throughput of 772 MB/s @ 1 GHz , when the shared buffer is located in the MSM, and we copied to the L2-SRAM of core 1 ('Virtual Core 0 to 1, MSM to L2'). If the shared buffer is located in the L2-SRAM of core 0 ('Virtual Core 0 to 1, L2 to L2'), the throughput we achieve is 45 MB/s @ 1 GHz. Currently we investigate why the copy between the L2-SRAM of the cores does provide so little throughput.

When utilising the experimental driver for the EDMA3 peripherals of the TI-C6678, and EDMA3 unit `EMDA3CC0`, we achieve a throughput of 4041 MB/s with a buffer size of 128 kB, transferred between two buffers in the L2-SRAM of core 0. The advantage of using the DMA unit over using the CPU for copying or movind data is that during the transfer the CPU can perform other tasks, thus the transfer happens in parallel to the processing.

3) Comparing Intel-SCC and TI-C6678: The best achieved throughput in single core measurements on the Intel-SCC was with a packet payload size of 4096 B where it achieved a throughput of 19.80 MB/s @ 533 MHz. The TI-C6678 achieved with the same packet payload size a throughput of 1148.52 MB/s @ 1 GHz.

For the distributed system version the Intel-SCC throughput is 33.57 MB/s @ 533 MHz with a packet payload size of 4096 B. The TI-C6678 achieves 512 MB/s @ 1 GHz.

V. CONCLUSIONS & FURTHER WORK

The first part of this paper introduced the two paradigms of OpenComRTOS, Interacting Entities, and Virtual Single Processor, and illustrated how they enable to develop truly distributed heterogeneous deeply embedded systems. Both paradigms enable it to build small systems as well as large systems without having to change the programming model at all. It is also possible to start with a small system and expand it over time if the need arises or the other way around. This is what is meant with the term scalability. Due to being build around the concept of packet switching the performance degradation caused by additional middleware layers are avoided in OpenComRTOS systems. This not only results in a better performance, but also in smaller memory requirements, and thus less power consumption. The architecture of OpenComRTOS is ideally suited for the multi/many cores systems such as the Intel-SCC and the TI-C6678, because it makes it very easy to use all processing power without having to worry about the details of the underlying hardware.

What has become clear in the performance measurements is that both the Intel-SCC and the TI-C6678 are complex architectures requiring a lot of attention to achieve best performance and predictable realtime behaviour. The developer must be very careful in placing data and code in memory and selecting the communication mechanism. In case of the Intel-SCC the access to the DDR3 memory has a very long latency with a minimum of 86 wait states, and is only available over the system wide shared routing network, which causes additional wait states. The approach taken in the TI-C6678 with a dedicated switching network (TeraNet) provides a much better throughput to the shared memory resources. Additionally, each core has it's own $512\ \mathrm{kB}$ of L2-SRAM which can be used to store code and local data, an approach not possible in case of the Intel-SCC. A local RAM of $512\ \mathrm{kB}$ might sound little but for OpenComRTOS it is more than sufficient, due to its small code size of around $5\ \mathrm{kB}$. This leaves in many cases sufficient space for user applications and device drivers.

The tests have also shown that shared memory presents some pitfalls, similar to the ones global variables represent in multi-threaded environments. Not only makes it the bus structure very complex, it also makes it very slow compared with the speed of the CPUs and it poses more safety and security risks, e.g. the cache must also be invalidated at the right time. Therefore, having large and local low wait state memory for each core with a fast dedicated communication network set up in a point-to-point topology with DMAs improves performance, and improves reliability when this memory can be marked as private to the core, thus preventing external cores from accessing and potentially corrupting it. This is an important issue for safety and security critical systems. Finally, multi/manycore designers should be aware that concurrency even on a single core combined with low latency is beneficial as it allows to reduce the grain size of the computations without suffering much overhead. It also increases throughput by overlapping computation with communication.

The communication infrastructure provided by the TI-C6678, with its packetisation and hardware-queue support, is similar to the internal architecture of OpenComRTOS, whereby all interactions are implemented using packet exchanges.

A. Further Work

While the basic port has been done, the integration into the OpenComRTOS ecosystem i.e. adapting the code generators and importing the multi-core topology as a library component is on-going. In parallel further optimisations are applied. Given the abundancy of hardware resources on modern multicore chips, research is focusing on dynamic resource scheduling, whereby a resource is not just CPU time but can also be any of the hardware capabilities. This is using an extended version of the distributed priority inheritance algorithm in OpenComRTOS.

ACKNOWLEDGEMENTS

The formal modelling of OpenComRTOS was partly funded under an IWT project of the Flemish Government in Belgium.

The Intel-SCC system we used for development was supplied by Intel Inc, in their data centre.

The TI-C6678 target hardware was provided by Thales.

REFERENCES

[1] Bernhard H.C. Sputh, Eric Verhulst, and Vitaliy Mezhuyev. OpenComRTOS: Formally developed RTOS for Heterogeneous Systems. In *Embedded World Conference 2010*, March 2010.

[2] E. Verhulst, R.T. Boute, J.M.S. Faria, B.H.C. Sputh, and V. Mezhuyev. *Formal Development of a Network-Centric RTOS*. Software Engineering for Reliable Embedded Systems. Springer, Amsterdam Netherlands, 2011.

[3] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[4] Eric Verhulst. Virtuoso: providing sub-microsecond context switching on dsps with a dedicated nanokernel. In *Proceeding of the International Conference on Signal Processing Applications and Technology, Santa Clara*, September 1993.

[5] Intel Labs. *The SCC Programmers Guide*, 2012. http://communities.intel.com/servlet/JiveServlet/downloadBody/5684-102-8-22523/SCCProgrammersGuide.pdf.

[6] Texas Instruments. *TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor (Rev. C)*. http://www.ti.com/litv/pdf/sprs691c.

[7] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[8] Bernhard H.C. Sputh, Eric Verhulst, Artem Barmin, and Vitaliy Mezhuyev. Safe Virtual Machine for C in less than 3 KiBytes. In *Embedded World Conference 2011*, March 2011.

[9] NXP. *NXP-CoolFlux Homepage*. http://www.coolflux.com/.

[10] The BareMichael framework: http://communities.intel.com/message/151910.

[11] Stefan Lankes, Pablo Reble, Carsten Clauss, and Oliver Sinnen. Shared Virtual Memory for the SCC. In Peter Troger & Andreas Polze, editor, *Proceedings of the 4th MARC Symposium*, Hasso Plattner Institute for Software Systems Engineering (HPI) in Potsdam, January 2012. Hasso Plattner Institute, University of Potsdam, Germany.

[12] Texas Instruments. *KeyStone Architecture Multicore Navigator*, September 2011. http://www.ti.com/lit/ug/sprugr9d/sprugr9d.pdf.

[13] Texas Instruments. *KeyStone Architecture Multicore Shared Memory Controller (MSMC)*, October 2011. http://www.ti.com/lit/ug/sprugw7a/sprugw7a.pdf.

 ISBN: 978-2-7257-0016-8

Efficient Implementation of the bare-metal Hypervisor MetalSVM for the SCC

Pablo Reble, Jacek Galowicz, Stefan Lankes, Thomas Bemmerl
Chair for Operating Systems, RWTH Aachen University
Kopernikusstr. 16, 52056 Aachen, Germany
{reble,galowicz,lankes,bemmerl}@lfbs.rwth-aachen.de

***Abstract*—The focus of this paper is the efficient implementation of our compact operating system kernel as a bare-metal hypervisor for the SCC. We describe source, functionality, and the operation of our kernel, as well as the interaction with the already published communication layer. Furthermore we give a detailed insight into the boot procedure of the SCC from reset to the starting point of our light-weight operating system kernel. This procedure is performed by a bare-metal framework, which is part of the MetalSVM project. Programmers can use our framework as a springboard for bare-metal programming on the SCC, which goes along with the first release of MetalSVM. Finally, we evaluate the performance of a paravirtualized Linux guest on the SCC hardware and present results of context switch latencies for Linux and MetalSVM hosts.**

I. INTRODUCTION

The Single-chip Cloud Computer (SCC) experimental processor [1] is a *concept vehicle* created by Intel Labs as a platform for many-core software research, which consists of 48 cores arranged in a 6×4 on-die mesh of tiles with two cores per tile. The intended programming approach for this cluster-on-chip platform is based on message passing [2].

For the parallelization of data-intensive algorithms, especially with irregular access pattern a shared memory programming model like *OpenMP* which is based on memory coherence offers an attractive and efficient alternative. If future many core processor architectures have to waive the memory coherency implementation in hardware, *MetalSVM* can enable shared memory programming on those architectures using virtualization.

One logical, but parallel and cache coherent virtual machine runs on top of a virtualization layer. With a Shared Virtual Memory (SVM) system this implements a classic approach for the realization of memory coherence in software in a bare-metal hypervisor. The virtualized Linux instance, called guest, will have the impression of being executed on a symmetric multiprocessor system. As a result, standard shared memory parallelized applications can run on future many-core platforms. Since the shared memory paradigm shows advantages in many scenarios, we are convinced that it is valuable to transparently provide memory coherence even on an architecture without according hardware support.

This paper is structured as follows: In Section II, we motivate the realization of *MetalSVM*[1] and summarize related work of our project. Afterwards, we present in Section III the structure and implementation details of the first version of *MetalSVM*. We describe the Boot process of the hypervisor kernel on the SCC platform in Section IV. Additionally, we compare context switch overhead and the hypervisor implementation performance between Linux and *MetalSVM* in Section V. In Section VI, we explain the benchmarks used for the evaluation of our kernel and present the respective performance results. The final Section VII summarizes this paper and gives an outlook to our next research goals.

[1] http://www.metalsvm.org

II. MOTIVATION AND RELATED WORK

Initially by forking *eduOS*, we started the further development of *MetalSVM*. *eduOS* is a very minimalistic operating system used for educational purposes at the RWTH Aachen University. It is inspired by *Unix* but does not aim to be fully POSIX compliant as, for instance, the Linux kernel or the MINIX kernel, which are also used for operating system courses and research [3].

In fact, the simplicity of *eduOS* leads to an easy customizability and tasks running in kernel space are executed near bare-metal. As a lightweight and small monolithic kernel, it provides adequate functionality for running user space programs. Figure 1 shows the basic kernel structures of *eduOS*.

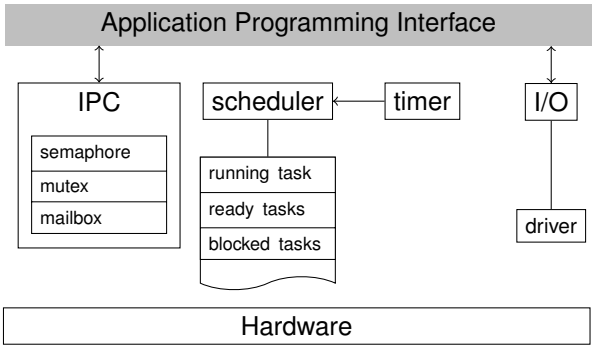


Fig. 1: Kernel structure of *eduOS*

MetalSVM, the further development of *eduOS*, represents a highly optimized codebase for running applications near bare-metal on the Intel SCC. Programmers can use our framework as a springboard for bare-metal programming on the SCC. In [4], we presented a first prototype, and in [5] further improvements of an SVM system, based on our framework. Here,

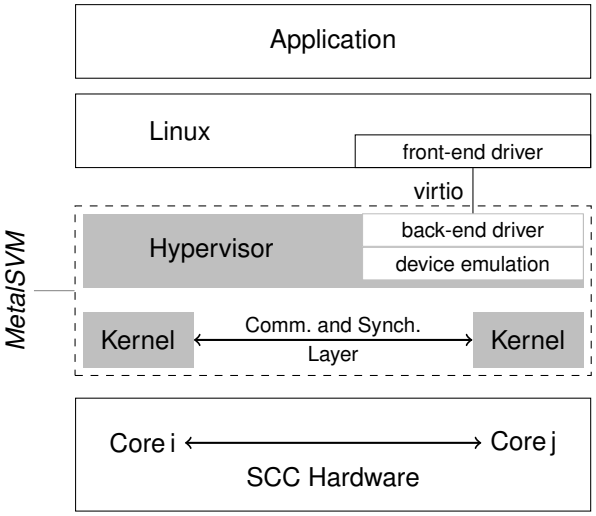


Fig. 2: Basic Concept of MetalSVM [4]

a shared memory application uses special SVM functions explicitly for shared memory allocation. A transparent use of the SVM layer by unchanged software will be enabled by a virtualization layer on top of the functionality of the *MetalSVM* kernel (see Figure 2).

From the application programmer's view, Linux user space applications have limited control over the preemption time, which is affected by context switching and interrupt handling. Consequently, this can be a good reason to run applications bare-metal to avoid this kind of overhead. However, one may be not interested or be able to take care of the rest of the necessary low-level work, which is the common reason for using an operating system. Since *MetalSVM* is configurable, the possibility exists to switch off infrastructure, for instance the hypervisor or the communication layer, which makes our framework comparable to bare-metal frameworks presented at the Intel Communities page [6], [7].

In [8], we evaluated the synchronization and communication hardware support of the SCC for inter kernel usage. For the integration of *iRCCE* into *MetalSVM*, this included an extension in the form of a mailbox system in combination with optimized synchronization support. The result is fast synchronous and asynchronous communication between user and kernel tasks of *MetalSVM* [9].

Besides *MetalSVM*, several projects handle the integration of an SVM system into virtual machines, for an easy application of common operating systems and development environments without changes. An example for such a hypervisor-based SVM system is *vNUMA* [10] that has been implemented on the Intel Itanium processor architecture. In [11] one founder of *vNUMA* proposed to extend this concept for Many-Core Chips. For x86-based compute clusters, the so-called *vSMP* architecture developed by ScaleMP[2] allows for cluster-wide cache-coherent memory sharing. This architecture implements a virtualization layer underneath the OS that handles distributed memory accesses via InfiniBand-based communication. In some respects, these approaches are similar to our hypervisor approach. Both implement the SVM system in an additional virtualization layer between the hardware and the operating system.

The main difference between these approaches is that *vSMP* and *vNUMA* explicitly use message-passing between the cluster nodes to transfer the content of the page frames, whereas our SVM system can cope with direct access to these page frames. In fact, we want to exploit the SVM system with SCC's distinguishing capabilities of transparent read/write access to the global off-die shared memory. This feature will help to overcome a drawback of other hypervisor-based approaches regarding fine granular operations. A recent evaluation [12] of ScaleMP's *vSMP* with synthetic kernel benchmarks as well as with real-world applications has shown that *vSMP* architecture can stand the test if its distinct NUMA characteristic is taken into account. Moreover, this evaluation reveals that fine granular operations such as synchronization are the big drawback of this kind of architectures. Our aim is to avoid this shortcoming by using the distinguished capabilities of transparent remote read/write memory on the SCC.

RockyVisor [13] is the name of another project for the realization of a hypervisor based symmetric multi-processing support for the SCC. In contrast to *MetalSVM*, this project targets the integration of its hypervisor into Linux and not on the base of a minimalistic kernel. Therefore, on the top of all Linux instances runs a virtualized Linux, which assumes that the SCC is an SMP system. From our point of view, such a *Linux on Linux* approach implies unneeded overhead.

III. KERNEL FEATURES

The intended usage for an SVM management system influences the hypervisor kernel. In this section, we detail the implementation of this monolithic kernel including interrupt handling, device drivers, file system, and hypervisor. Additionally, we give reasons for specific design decisions by concrete applications.

The focus in this paper is the kernel implementation for the SCC. However, we compare this implementation to different hardware architectures supported by *MetalSVM*, whose concept is divided in a hardware dependent and independent part.

A. Hypervisor

The fact that a guest kernel is aware that it runs as a guest and uses hypercalls to do privileged operations is called paravirtualization [14]. Using an existing hypervisor solution from the Linux kernel has been the first choice for the integration into *MetalSVM* [15]. This way we can avoid changes on the Linux kernel code, since interaction between host and guest is based on a de facto standard virtualization interface. *lguest* is an appropriate match in this context, because its about 5000 lines of code keep it quite simple. Despite its small

[2]http://www.scalemp.com

size it provides all required features for the realization of the *MetalSVM* project [16].

For development and testing purposes, we use *QEMU*[3], which is a generic and open source machine emulator and virtualizer. To simplify our tests of standard kernel components, we integrated a driver for the Realtek RTL8139 network chip, which is also supported by *QEMU* as an emulated device.

B. Device Drivers

Communication between the SCC cores running *MetalSVM* is not limited to the iRCCE library and its mailbox extension. With the integration of *lwIP*, a light-weight TCP/IP library, the flexibility is increased [17]. Consequently, BSD sockets are made available to user space applications to establish communication between the SCC cores and the MCPC. In [4], we demonstrated the convincing performance of the resulting network layer.

The network capabilities besides other devices of *MetalSVM* will be forwarded to the guest operating system through the hypervisor via *virtio*. Virtio is Rusty Russell's draft to create an efficient and well-maintained framework for IO-virtualization of virtual devices commonly used by different hypervisors [18]. In our scenario, for instance the network capabilities of *MetalSVM* are used as a backend by just forwarding the requests of the Linux guest operating system to the hypervisor.

C. Interrupt Management

The SCC platform includes 48 P54C cores. As a second generation of *Pentium* cores, the P54C is the first processor which is based on an on-chip local *Advanced Programmable Interrupt Controller* (APIC). This local APIC is used to program the local timer interrupt, which can be used to trigger the scheduler periodically. *MetalSVM* uses a simple priority-based round-robin scheduler, described in detail in Section V.

Beside the timer interrupt, the local APIC possesses two programmable local interrupts (*LINT0* and *LINT1*). Interrupts achieve an important role, because the SCC does not use the traditional way to integrate I/O devices (*IO-APIC*) or to send inter-processor interrupts (*IPIs*). Therefore, a core configuration register exists for each core of the SCC, which is mapped to the address space of all cores. A special bit in these registers triggers a *LINT0* or a *LINT1*. As a result, core x is enabled by the memory-mapped configuration registers to trigger an interrupt on core y. However, with this mechanism the receiving core is now able to determine the origin of the interrupt.

The update of Intel *sccKit* to version 1.4.0 includes a *Global Interrupt Controller* (GIC), which provides a more flexible way to handle interrupts [19]. If an interrupt is triggered by the GIC, the receiver is able to determine the origin of this interrupt. *MetalSVM* uses the GIC especially for inter-core communication via iRCCE or our mailbox system [5]. Here, the information about the origin of an interrupt increases the scalability.

[3]http://www.qemu.org/

D. File system

Since the SCC provides no non-volatile storage, a file system is physically limited in use. Nevertheless, *MetalSVM* has an elementary *inode* file system with an initial population loaded from a ramdisk file. This file system can be manipulated at runtime.

The integration of *newlib*[4], which is a C library intended for use on embedded systems, extends the usage of *MetalSVM*. Regarding the mode to run user-space applications on *MetalSVM* arises the possibility to access custom character devices by the provided `/dev` directory. These can be implemented very comfortably using a well defined interface.

IV. BOOT ON THE SCC

MetalSVM is *Multiboot*[5] compliant. This means that the project framework creates an ELF kernel file and an initial ramdisk image file. A boot loader like GRUB can easily use these files to boot *MetalSVM* on commodity x86 hardware.

Because the available SCC hardware is a research prototype, the booting process differs from commodity hardware. Differences to commodity hardware are the absence of BIOS support and the lack of stand-alone memory initialization of this experimental platform. The only possibility to bootstrap the SCC cores is preloading their memory content into a bootable state. Thus, the general system initialization is realized by a standard PC (MCPC) with direct access to the memory of the SCC and its configuration registers.

In the following, we describe the function of our framework to bring the SCC Platform into a *Multiboot* compliant state. As a result, an entry point for our 32 bit minimalist *Multiboot* kernel is created. Additionally, we describe the interaction with the common *sccKit* tools to boot up the SCC platform with *MetalSVM*.

Initially, the boot procedure starts by pulling the reset pins of the SCC cores. Next, its Lookup Tables (LUT) are initialized and the memory is set into a bootable state for each core. After a reset pin release the instruction pointer of each core holds the hardwired address `0xfffffff0`. As the SCC does not provide any form of boot loader, our framework provides minimal assembler code for this purpose, which needs to be located at this position. Starting the operation in real mode, this code initializes the stack pointer, installs a rudimentary GDT, switches the processor to protected mode and subsequently to 32 bit mode. As a last step, this setup procedure jumps to the alignment value of the *MetalSVM* kernel, address `0x100000`.

The compiler has to support the pentium architecture for the generation of ELF format output of our minimalist kernel for the SCC. ELF, as the standard binary file format for Unix systems, is currently not supported by the *sccKit* tools. Therefore, the utility `objcopy` is used to generate a directly loadable, raw binary kernel file by discarding all symbols and relocation information. The previously described startup

[4]http://sourceware.org/newlib/

[5]http://www.gnu.org/software/grub/manual/multiboot/

routine, from real to protected mode, a data struct, containing information which is generally provided by the bootloader, and the kernel itself are composed to a single image by `sccMerge`.

Next, `sccMerge` creates rules for the configuration of the LUTs and one object file per memory controller of the SCC platform. Subsequently, `sccBoot` loads the generated object files into the off-die memory and finally `sccReset` is used to release the reset pins of the SCC cores.

V. SCHEDULING

Requirements to a scheduler of the presented hypervisor are lower compared to schedulers of popular modern operating systems. Specifically, the intended use for the scheduler is to handle a few tasks, such as the guest kernel, daemon and monitoring tasks. Hence, a simple but fast algorithm is applied to manage tasks. The scheduler keeps an array with as many items as priority steps exist. Per priority there is one linked list of tasks waiting for execution. Between two timeslices the scheduler appends the previous task to the end of its priority list and selects the head of the current processed priority level list for execution.

The small set of implemented priorities in *MetalSVM* provides the possibility to apply optimization. One optimization is already implemented in the networking layer. Network packet traffic is handled in a special kernel task whose priority can be changed. This way it is possible to balance between high network throughput and overall system latency.

Version 0.1 of *MetalSVM* supports 32 different priority levels. This small number allows *MetalSVM* to create a bitmap of used priority queues in one 32 bit integer. Consequently, with one assembler instruction (`msb`) is it possible to determine the highest used priority queue, which promises an extremely low overhead. Before leaving any interrupt handler, the handler checks, if a task with higher priority is able to run and calls the scheduler if required. In our scenario, a reduction of the latency of the network stack can be achieved, by holding the network thread at a higher priority than the computation tasks.

A. Hardware Context Switch

Early versions of *MetalSVM* used the x86 hardware task switching by default for a context switch. Here, a context switch is performed by a `JMP` to a *TSS* descriptor, which has the advantage of a very simple application. Therefore, the *TSS* (Task State Segment), which stores the state of a task, except the FPU state, is restored.

A disadvantage of this method is the lack of selection which registers are saved and restored [20]. Furthermore, the number of *TSS* entries in the segment table is limited to 8192 [21]. Due to portability reasons most modern operating system implementations use software task switching.

However, a basic component of the SCC is the classic P54C core, which could result in an increased scheduling performance of hardware context switching. Besides a benchmark of the hypervisor layer, we evaluate this assumption in the next section.

VI. BENCHMARKS

Benchmark results of different subsystems of *MetalSVM*, excluding the kernel, have been already published. An evaluation of the synchronization support, including different spin lock and barrier implementations is presented in [8]. In [4], the network layer and a first prototype of the SVM layer are evaluated. Further optimizations of the SVM layer and the mailbox extension of iRCCE are presented in [5].

In this section of this paper, we analyze advantages of a bare-metal implementation of *MetalSVM*. We compare the context switch overhead of *sccLinux 2.6.38* to *MetalSVM* 0.1 on the SCC platform. Additionally, we compare the *lguest* implementation of *MetalSVM* to the implementation of the Linux kernel 2.6.32 and 2.6.38.3. For a comparison of the results, the benchmark application is the single process running on *sccLinux*.

For the benchmark in this section, we obtained measurements by running a single instance of the selected host operating system on a single core of the SCC platform[6]. Because, *sccLinux* in a version 2.6.32 is currently not available in a configuration with *lguest* support, we used an Intel Celeron 550 test system with a frequency of 2 GHz, to benchmark the context switch latencies.

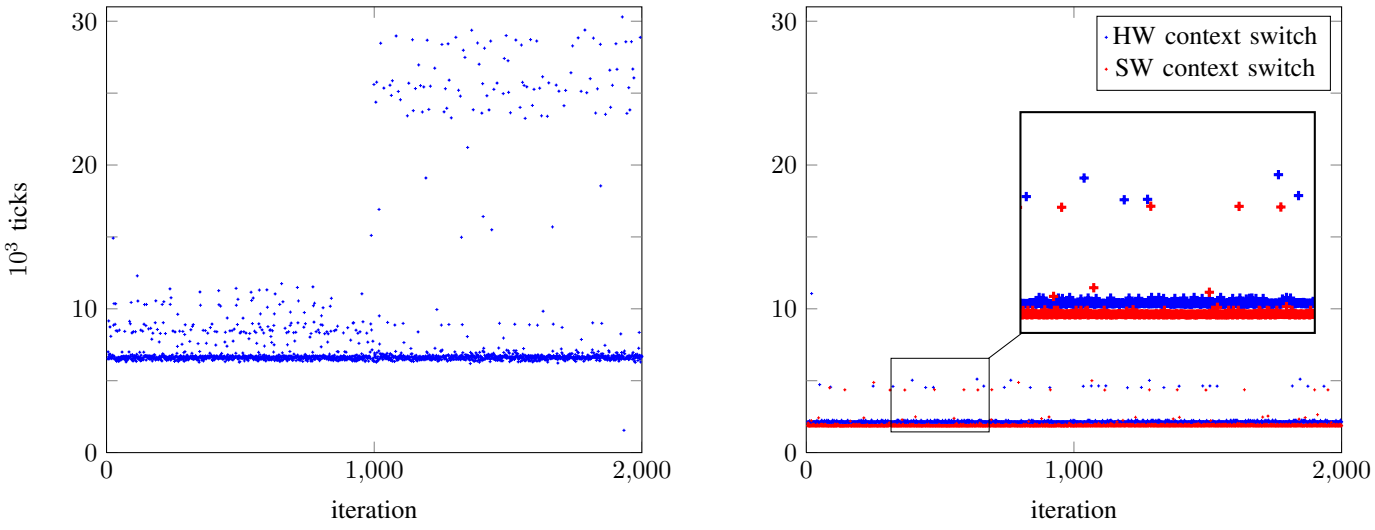
A. Context Switch Latency

For the measurement of context switch latency, two tasks are running on a single core with a high priority. Each task periodically reads the time stamp counter in a loop and stores the result at a shared memory location. Measured gaps, which are shorter than a timeslice and longer than an iteration without interruption, are recorded as an indicator for the latency of a context switch and visualized as a scattered plot in Figure 3. This method is comparable to the hourglass benchmark [22]. But in contrast to our benchmark, the hourglass benchmark tests the general preemption time and gives no information about the context switch latency.

Thus, the benchmark results from Figure 3 can be used for a comparison of context switch latencies between *sccLinux* and *MetalSVM*. As reported by Figure 3a, *sccLinux* has a minimal context switch overhead of about 6400 processor cycles. Figure 3a indicates a certain noise, which has no clear signature and changes from time to time. The picture is different for *MetalSVM*, which generates a minimal context switch overhead of 2100 processor cycles. This is more than 3 times faster. However, Figure 3b shows a second level of about 5000 ticks for context switch latencies. This effect is caused by the process of the lwIP driver, which is running with a high priority.

The scale-up from Figure 3b visualizes the differences between hardware and software context switch for *MetalSVM* on the SCC platform. Here, no significant effect of the context switch method to the context switch latency, except a constant offset, can be identified.

[6] core/mesh/memory frequency: 533 MHz/800 MHz/800 MHz



(a) sccLinux 2.6.38 (b) *MetalSVM*

Fig. 3: Context switch latencies on the SCC (2000 iterations)

B. Hypervisor Performance

The hypervisor plays an important role to establish a transparent shared virtual memory environment. Obviously, its overhead has a significant impact on the performance of its guest machine, for instance concerning memory management, context switches and process handling.

Measurements of three representative latencies identify a reduced virtualization overhead of *lguest* in combination with *MetalSVM*. The *context switch* from guest execution to host execution is performed at each hypercall and at the majority of interrupts. *Page faults* in a guest application can involve up to 3 guest-host roundtrips. Therefore, a fast resolving is aimed for. We measured the duration of *system calls*, exemplary for `getpid`, `fork`, `vfork`, and `pthread_create`. Here, `getpid` indicates the overhead of a system call, since its payload execution time is very low. Due to optimizations in interrupt delivery, `getpid` does not involve a host-guest context switch. The difference of 400 ticks between Linux and MetalSVM as the host operating system can be explained by cache effects. `fork` and `vfork` are used to show the amount of ticks needed for the creation of a task and the copy operation of a whole page directory of the original task. A huge difference between Linux and MetalSVM for the execution time of `pthread_create` is noticeable. This effect can be explained by the coarse granularity of the current timer implementation of *MetalSVM*. Here, the processor frequency has a direct impact.

As a *real-life example* we used a floating point operation intensive application in the form of the jacobi solver algorithm. We measured the overall execution efficiency within the virtual guest machine. Additionally, a second setup indicates the overhead of a task plus floating point context switch by running two instances of the solver.

TABLE I: Benchmark results for the Intel Celeron platform (Linux 2.6.32)

Benchmark	Hypervisor		Ratio $\frac{MSVM}{Linux}$
	Linux	*MetalSVM*	
Host-guest context switch	1 406	1 347	96 %
Page fault	40 426	31 978	79 %
`getpid()`	1 039	626	60 %
`fork()`	446 224	301 831	68 %
`vfork()`	163 421	117 536	72 %
`pthread_create()`	3 678 968	40 022 838	1 088 %
Jacobi solver (128x128 Matrix)	$156 \cdot 10^9$	$99 \cdot 10^9$	63 %
Jacobi solver (2 instances)	$317 \cdot 10^9$	$199 \cdot 10^9$	63 %

Values in processor ticks

The 3 tables (I, II, and III) show the tick count of both hypervisor implementations, Linux and *MetalSVM*, for different stages of the development. The light weight *MetalSVM* kernel results in a successful reduction of overhead for our implementation in combination with memory handling code optimizations of the hypervisor (cf. Table I). However, these measurements were taken at an earlier development stage of the hypervisor.

Table II shows benchmark results of *MetalSVM* version 0.1 and a more recent Linux kernel (2.6.38.3), which is available with *sccKit* 1.4.1 for the SCC platform. The Linux kernel has undergone performance improvements from version 2.6.32 to 2.6.38.3, which affects the benchmark results. However, we see a major advantage of a light weight solution, concerning customizability and transparent performance analysis.

TABLE II: Benchmark results for the Intel SCC platform (Linux 2.6.38.3)

Benchmark	Hypervisor		Ratio $\frac{MSVM}{Linux}$
	Linux	*MetalSVM*	
Host-guest context switch	2 042	2 113	103 %
Page fault	918 679	867 676	94 %
`getpid()`	191	191	100 %
`fork()`	3 216 767	3 101 387	96 %
`vfork()`	220 317	236 207	107 %
`pthread_create()`	16 256 988	10 883 839	67 %
Jacobi solver (32x32 Matrix)	$3.74 \cdot 10^9$	$3.74 \cdot 10^9$	98 %
Jacobi solver (2 instances)	$7.51 \cdot 10^9$	$7.48 \cdot 10^9$	98 %

Values in processor ticks

TABLE III: Benchmark results for the Intel Celeron platform (Linux 2.6.38.3)

Benchmark	Hypervisor		Ratio $\frac{MSVM}{Linux}$
	Linux	*MetalSVM*	
Host-guest context switch	3 020	2 590	86 %
Page fault	40 388	43 985	109 %
`getpid()`	607	595	98 %
`fork()`	351 907	371 381	106 %
`vfork()`	132 142	137 366	104 %
`pthread_create()`	1 020 630	40 049 784	3924 %
Jacobi solver (32x32 Matrix)	$2.04 \cdot 109$	$2.03 \cdot 109$	99 %
Jacobi solver (2 instances)	$4.08 \cdot 109$	$4.13 \cdot 109$	101 %

Values in processor ticks

VII. CONCLUSION AND OUTLOOK

In this paper we presented a bare-metal hypervisor, with the roots of a Unix-like monolithic kernel, used for educational purposes. Our framework extends the software package *sccKit* of the many-core platform to run our configurable light-weight bare-metal programming environment. Performance evaluation of the context switch latency proves the assumption that kernel tasks can be executed close to bare-metal. Thus, broad functionality like interrupt handling and inter core communication in a synchronous as well as asynchronous manner is provided.

This meets the requirements for the integration of an SVM system perfectly, which we have already shown in [4] by using an adapted shared memory application. Here, the light-weight kernel benefits from the efficiency of its subsystems.

The benchmark results of selected system calls for a Linux guest system underline the potential of a bare-metal hypervisor implementation. Considered as a whole, it features a convenient development base for research due to its simplicity and limited base of supported hardware architectures.

For transparent execution of shared memory parallelized applications, we plan to boot and connect multiple instances of the presented kernel and run a single paravirtualized Linux instance on top of the hypervisor layer.

ACKNOWLEDGMENT

The research and development was funded by Intel Corporation. The authors would like to thank especially Ulrich Hoffmann, Michael Konow and Michael Riepen of Intel Braunschweig for their help and guidance.

REFERENCES

[1] *SCC External Architecture Specification (EAS)*, Intel Corporation, November 2010, Revision 1.1. [Online]. Available: http://communities.intel.com/docs/DOC-5852

[2] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and improvements of programming models for the intel scc many-core processor," in *Proceedings of the 2011 International Conference on High Performance Computing and Simulation (HPCS 2011)*, Istanbul, Turkey, July 2011, pp. 525 –532. [Online]. Available: http://dx.doi.org/10.1109/HPCSim.2011.5999870

[3] A. Tanenbaum and A. Woodhull, *Operating Systems: Design and Implementation*, 3rd ed. Prentice Hall, 1997.

[4] S. Lankes, P. Reble, C. Clauss, and O. Sinnen, "The Path to MetalSVM: Shared Virtual Memory for the SCC," in *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium*, Potsdam, Germany, December 2011. [Online]. Available: http://communities.intel.com/docs/DOC-19214

[5] S. Lankes, P. Reble, C. Clauss, and O. Sinnen, "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 (PMAM 2012) in conjunction with the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2012)*, New Orleans, LA, USA, February 2012. [Online]. Available: http://doi.acm.org/10.1145/2141702.2141708

[6] ET International, "ETI's SCC Development Framework available," August 2011. [Online]. Available: http://communities.intel.com/thread/17643

[7] M. Ziwisky, "BareMichael baremetal framework," April 2012. [Online]. Available: http://communities.intel.com/thread/28001

[8] P. Reble, S. Lankes, C. Clauss, and T. Bemmerl, "A Fast Inter-Kernel Communication and Synchronization layer for MetalSVM," in *Proceedings of the 3rd MARC Symposium, KIT Scientific Publishing*, Ettlingen, Germany, July 2011. [Online]. Available: http://communities.intel.com/docs/DOC-6871

[9] C. Clauss, S. Lankes, T. Bemmerl, J. Galowicz, and S. Pickartz, *iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer*, Chair for Operating Systems, RWTH Aachen University, July 2011, Users' Guide and API Manual. [Online]. Available: http://communities.intel.com/docs/DOC-6003

[10] M. Chapman and G. Heiser, "vNUMA: A Virtual Shared-Memory Multiprocessor," in *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, USA, Jun 2009, pp. 349–362.

[11] G. Heiser, "Many-Core Chips — A Case for Virtual Shared Memory," in *Proceedings of the 2nd Workshop on Managed Many-Core Systems (MMCS)*, Washington, DC, USA, March 2009, p. 4 pages.

[12] D. Schmidl, C. Terboven, A. Wolf, D. an Mey, and C. Bischof, "How to Scale Nested OpenMP Applications on the ScaleMP vSMP Architecture," in *Proceedings of 2010 IEEE International Conference on Cluster Computing*, September 2010, pp. 29 –37.

[13] J.-A. Sobania, P. Tröger, and A. Polze, "Towards Symmetric Multi-Processing Support for Operating Systems on the SCC," in *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium*, Potsdam, Germany, December 2011.

[14] A. Whitaker, M. Shaw, and S. D. Gribble, "Denali: Lightweight Virtual Machines for Distributed and Networked Applications," in *Proceedings of the USENIX Annual Technical Conference*, 2002.

[15] S. Lankes, "First Experiences with SCC and a Comparison with Established Architectures," Invited talk at the 1st MARC Symposium, Braunschweig, Germany, November 2010. [Online]. Available: http://communities.intel.com/docs/DOC-5848

[16] R. Russell, "lguest: Implementing the little Linux hypervisor," in *Proceedings of the Linux Symposium, Ottawa, Canada*, 2007.

[17] A. Dunkels, *Design and Implementation of the lwIP TCP/IP Stack*, Swedish Institute of Computer Science, 2001.

[18] R. Russell, "virtio: Towards a De-Facto Standard for Virtual I/O Devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 95–103, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1400097.1400108

[19] *The sccKit 1.4.x User's Guide*, Intel Labs, October 2011. [Online]. Available: http://communities.intel.com/docs/DOC-6241

[20] D. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed., A. Oram, Ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002.

[21] *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A*, Intel Corporation, August 2007.

[22] J. Regehr, "Inferring Scheduling Behavior with Hourglass," in *Proceedings of the USENIX Annual Technical Conference FREENIX Tracki*, Monterey, CA, USA, June 2002, pp. 143–156.

 ISBN: 978-2-7257-0016-8

BareMichael: A Minimalistic Bare-metal Framework for the Intel SCC

Michael Ziwisky
Department of Electrical and Computer Engineering
Marquette University
Milwaukee, WI 53233
Email: michael.ziwisky@mu.edu

Dennis Brylow
Department of Mathematics, Statistics, and Computer Science
Marquette University
Milwaukee, WI 53233
Email: brylow@mscs.mu.edu

***Abstract*—The many-core Intel SCC processor is one of a class of emerging, highly parallel computer architectures. Intel provides a modern Linux kernel which, running on the SCC as a separate instance per core, is able to load and launch user applications. However, there is a lack of open-source tools to facilitate development of "bare-metal" SCC applications – applications that are run directly on the chip without the support, overhead, or restrictiveness of an underlying operating system.**

To help fill this void, we present BareMichael, a minimalistic framework for compiling, loading, and launching mixed C and assembly code on the bare-metal Intel SCC. The framework also includes MikeTerm, a one-way pseudo-terminal for displaying output from an application as it executes on the chip. We share our solution in the hope that it will lower the barrier for others to begin development in a bare-metal environment on the SCC. Furthermore, we demonstrate the utility of BareMichael through two applications: supporting the use of the RCCE message-passing library, and serving as the foundation for a port of the Embedded Xinu operating system.

I. INTRODUCTION

The Single-Chip Cloud Computer (SCC) experimental processor is a "concept vehicle" created by Intel Labs as a platform for many-core software research [1], [2]. It features 48 processing cores based on the P54C architecture and a 256 Gb/s bisection bandwidth mesh network-on-chip (NoC). The chip is organized into 24 tiles, each of which contains two cores, a router, and 16 kB of shared memory that is accessible to all cores via the NoC. This fast, on-chip memory is referred to as the "message-passing buffer" (MPB).

Intel provides support software for SCC development including *SCC Linux*, a modern Linux kernel, and *sccKit*, a suite of tools for interacting with the chip via an attached "management console PC" (MCPC). While the environment of SCC Linux offers many convenient features, such as access to common Linux system services and the ability to interact with cores via an `ssh` session, it is also a restrictive environment, forcing developers to either run their SCC applications within a low CPU privilege level, or to modify the kernel itself to enable more advanced functionality.

It is thus desirable to be able to run applications in a "bare-metal" environment with neither the support nor the overhead and restrictions of an operating system. However, the barrier to get bare-metal C code running on the SCC and to get feedback from its execution is a significant one. We have overcome this barrier, and we share our solution, BareMichael, in the hopes that it will spare others the tedium and difficulty of coding the initialization and support routines necessary to begin development of bare-metal SCC applications.

The BareMichael framework enables a developer to execute bare-metal code on the SCC with supervisor-level access to all aspects of the chip. The framework is lightweight, minimalistic, and open-source. In the remainder of this paper, we describe the framework's platform initialization process, list the tools upon which it relies, describe a couple of applications for which we have used the framework, and discuss the other offerings for bare-metal SCC development.

II. THE BAREMICHAEL FRAMEWORK

BareMichael is a minimalistic framework to support bare-metal programming on the SCC. It is primarily a boot loader, not an operating system. Thus, it does not provide operating system functionality, but it may serve as a foundation upon which an operating system (or any other program) may be built. Along with a series of routines for initial configuration of an SCC core, BareMichael is packaged with `libxc`, a subset of the standard C library originally implemented for the Embedded Xinu kernel [3]. Upon the framework, developers may implement bare-metal code in C, x86 assembly, or a combination of the two. The framework also includes some SCC-specific helper functions and definitions to do things like reading the local core ID, reading mesh and tile clock frequencies, addressing MPBs and configuration registers, acquiring and releasing tile lock registers, and triggering inter-core interrupts. As BareMichael is an open-source tool, the implementations of all of these functions are exposed to the developer who is free to modify, remove, or reimplement them at will.

A. Platform Initialization

The following is a brief walkthrough of the code path BareMichael steps through to initialize an SCC core. This description, accurate for the latest versions (4, 5, 6, and 7) of the framework, illuminates the BareMichael startup process so that a developer may understand both how it works and

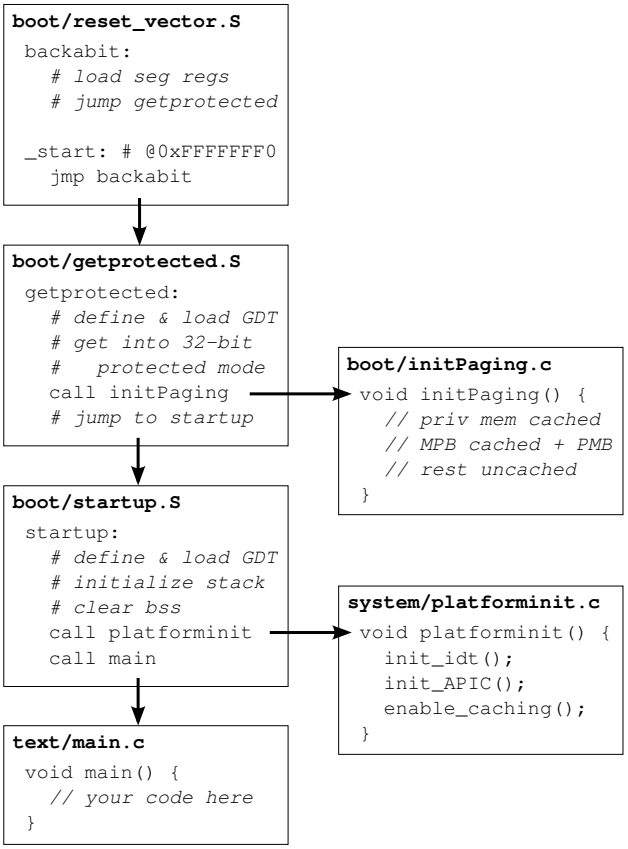


Fig. 1. Per-core initialization procedure of BareMichael.

how it may be modified to suit particular needs. Paragraph headers identify the location of the code being discussed, and a schematic representation of the entire process is illustrated in Figure 1.

a) boot/reset_vector.S: Based on the Intel P54C, each SCC core boots in "real mode," and consequently has access to just a 20-bit address space. In spite of this limitation, the first instruction a core executes after its reset pin is released is loaded from memory address 0xFFFFFFF0, sixteen bytes from the end of a 32-bit address space. We put a short relative jump instruction here, which takes us back just far enough to initialize the core's segment registers and stack pointer, then far-jump down to a `getprotected()` routine located within the first mebibyte of memory.

b) boot/getprotected.S: The `getprotected()` routine takes the processor into 32-bit "protected mode" by setting up the necessary CPU configuration data structures and registers, including a global descriptor table (GDT) to define flat code and data segments. Then a page table is created for virtual memory management.

c) boot/initPaging.c: The default look-up table (LUT) for an SCC core, which maps core addresses into a larger system address space, splits the core's address space into sections including private memory, shared memory, message passing buffer space, and configuration register space. Our page table flatly maps all of this space with cache disabled for all but private RAM and message passing buffers. Message passing buffer pages also have the PMB flag set to enable special caching features of the SCC [2]. With the page table configured and enabled, the core jumps to the `startup()` routine.

d) boot/startup.S: The `startup()` code gets linked together with `libxc` and the rest of the developer's bare-metal code to create the main image, which may be located in private memory wherever the developer chooses (specified via a Makefile variable). The `startup()` routine defines and loads a new (but identical) GDT within the main image to allow for easier addressing of the data structure should the developer wish to access it later. Space then is allocated for an interrupt descriptor table (IDT) which will be loaded with descriptors momentarily. After initializing a stack, clearing the `bss` section of the image, and initializing the floating point unit, the core calls `platforminit()`.

e) system/platforminit.c: Among the duties of the `platforminit()` routine are calls to initialize and enable the local advanced programmable interrupt controller (APIC), load the IDT with some default descriptors, and enable caching. As of version 3, the framework includes real-time clock support using the local APIC timer. If this feature is enabled (via a definition in `include/conf.h`), its initialization function is called here. Interrupt vectors 0x00 through 0x1F are reserved for CPU faults and exceptions, and the default handlers BareMichael assigns to these vectors print out information about the state that the system was in when the interrupt occurred. Such information is useful for debugging. After `platforminit()` returns, BareMichael calls the `main()` function in `text/main.c`, which is assumed to be the starting point of the developer's code.

To summarize, we now describe the state of an SCC core after BareMichael initialization. The setup routine brings the SCC core to 32-bit protected mode at privilege level 0 (supervisor level). Virtual memory management is enabled with page table entries present only for the core addresses that are mapped to actual system addresses by the default LUT configuration. Private memory is configured to have cache enabled, MPB-mapped pages have cache enabled and the SCC-specific PMB flag set, and all other sections have cache disabled. The local APIC is enabled and, by default, its periodic timer is set up to trigger a handler (found in `system/clock.c`) every millisecond. If the framework is configured for RCCE support (see Section III), the core's MPB space is initialized to zeros and a heap is initialized to allow dynamic management of private memory.

B. MikeTerm

BareMichael applications can print text back to the MCPC through a call to `printf()`. This function simply writes data to a circular buffer in memory where it can be seen and retrieved by the MCPC via the SCC's system interface. Each core has a different buffer allocated for this purpose. Running

 ISBN: 978-2-7257-0016-8

```
[00]: Hello, World -- I'm core 0!

[01]: Hello, World -- I'm core 1!

[05]: Hello, World -- I'm core 5!

[24]: Hello, World -- I'm core 24!

[47]: Hello, World -- I'm core 47!

[00]: I'm going to trigger core 47's LINT0 now.

[47]: I've been interrupted!
[47]: (SCC has been booted for 2 seconds)

[00]: Now I'm toggling core 47's LINT1.

[47]: Another interruption!
[47]: (SCC has been booted for 5 seconds)
^C
Thanks for flying MikeTerm!
```

Fig. 2. Sample output from MikeTerm. In this sample program, each booted core says "Hello." Then, after a short delay, core 0 toggles each of core 47's APIC interrupt pins with a delay in between. Core 47 has set these interrupt vectors to point to handlers that print out the total time passed since boot up. That time is kept track of by the real-time clock which operates based on the APIC timer and the tile clock frequency.

on the MCPC, a utility called *MikeTerm* acts as a one-way pseudo terminal, periodically polling each of the 48 buffers and printing any text found therein. All output from MikeTerm is preceded by a core identifier. Because MikeTerm scans the shared memory buffers sequentially, it is not guaranteed that its output will be printed in the order in which the cores wrote to their respective buffers. The output from any given core will be delivered in the order in which the core printed it, but ordering of output between any two cores is not necessarily preserved. Additionally, if a core is writing to its buffer faster than MikeTerm is retrieving it, old data will be overwritten and lost without being printed. No protections are built in to prevent this. The default configuration of the framework allocates 64 KiB buffers which get polled by MikeTerm roughly once per second, so data is likely to be lost when output rates are greater than about 64,000 characters per second. BareMichael currently offers no mechanism for interacting with running SCC programs by feeding data in the other direction, from the MCPC to the chip.

C. Build Environment and Dependencies

1) Dependencies: BareMichael leverages some open-source utilities for image compilation, image loading, and delivering output through MikeTerm. The framework uses the `i386-unknown-linux-gnu` cross-compiler tools from gcc version 3.4.5 to produce flat binary object files. *sccKit* is a suite of utilities, provided by Intel, that run on the MCPC and interact with the SCC. BareMichael is compatible with sccKit version 1.4.1, and it uses the `bin2obj`, `sccMerge`, `sccBoot`, and `sccReset` tools for loading binaries into SCC memory and toggling reset pins of individual cores. MikeTerm uses `sccDump` and `sccWrite` to access print buffers in shared memory.

2) Compilation and Execution: Compilation of both MikeTerm and the SCC image is managed using Makefiles written for the GNU `make` utility. MikeTerm is written in C++ and located in the `miketerm` directory. To compile it, simply change to that directory and invoke `make`.

BareMichael expects the directory containing sccKit binaries to be included in the user's `PATH` environment variable. Paths to the cross-compiler and `bin2obj` tool must be specified in the framework's Makefile, which is located at `compile/Makefile`. The Makefile also includes a configuration variable for specifying a list of cores to boot. After defining these few variables, compiling and running a bare-metal application is very simple and straightforward. The default `make` target builds the image; the `run` target loads that image into SCC memory and releases the resets of the specified cores. The `main()` function in `test/main.c` is the entry point for the developer's code, and if all of the developer's code is contained in that file (or in any set of files already in the framework), a simple '`make; make run`' is all that is needed to get the code running on the SCC. Follow it up with '`../miketerm/miketerm`' to view output from the cores. If additional source files need to be linked, one must add them to one of two lists in the Makefile: C source files get added to the `C_FILES` list, while assembly files belong in the `S_FILES` list.

3) Advanced Capabilities: Though most developers probably will be satisfied with the default configuration of the build environment, additional customization is possible. One simple example is changing the memory address to which the main image gets loaded onto the core. This is easy to modify as it is already defined by a variable (`IMG_ADDR`) in the Makefile. However, the framework has other potential capabilities – such as loading and booting different images on different cores – that are possible to realize but not as simple to exploit. For this reason, we disclose the roles of a few files that the build process creates along the way to creating a loadable SCC image.

Initially, the source is compiled into three flat binary object files: the reset vector, the "get protected" and paging initialization code, and the main image. The file `compile/load.map` is created and populated with the names of these three objects, each preceded by the memory address (32-bit core address, not a memory controller address) to which it is to be loaded. This file serves as the input to the `bin2obj` tool, which creates a text file, `compile/battle.obj`, that represents a composite of the three objects. The `sccMerge` tool decides where to load the composite image into SCC memory and how to set initial core LUT configurations. The tool makes these decisions based on three arguments: the number of cores to be served by each memory controller (12 by default), the size per memory controller in GiB (8 by default), and the contents of a `.mt` input file. BareMichael creates the file `compile/battle.mt`

and populates it with 48 lines, each of which identifies a core, a memory controller, a "memory slot" (between 0 and 47, inclusive), and a `.obj` file. By default, this file assigns to each core: the nearest memory controller; a memory slot between 0 and 12, which is assigned in increasing numerical order to the 12 cores sharing a memory controller; and the object file that was built earlier, `compile/battle.obj`. The output of `sccMerge` is a directory, `compile/obj/`, and files therein that define the SCC memory contents and the LUT configurations. This directory is provided as an argument to `sccBoot`, which does the actual loading of SCC memory and configuring of LUTs. Finally, the framework issues the command `'sccReset -r <list of cores>'` to release the reset pins of the desired cores.

Clearly, the build procedure may be altered in a few ways – most notably through modifications to the `.mt` file – to customize how SCC memory gets loaded and distributed among cores. As an example, one may arbitrarily assign `.obj` files to cores in the `.mt` file to boot heterogeneous images among the cores. Of course, this requires building multiple `.obj` images, so multiple load maps must be defined and fed to `bin2obj`. Implementation of such alterations is left to the interested developer.

III. Integration with RCCE

RCCE [4] is a message-passing software library that Intel Labs designed and implemented in conjunction with the SCC hardware. The current version of the library, V2.0, may be compiled for use in SCC Linux, a kernel port also supplied by Intel, or for use in a bare-metal environment. However, because bare-metal RCCE is a library and not an environment itself, it does not provide the execution framework needed to run bare-metal applications on its own. In addition to a CPU initialization process, RCCE demands:

- POSIX functions `mmap()` and `munmap()` for virtual memory management,
- file operations such as `open()`, `flush()`, and `fprintf()`,
- `malloc()` and `free()` for dynamic memory management, and
- various additional C library functions.

These gaps are filled by the v6 release of BareMichael, allowing the developer to use the unmodified bare-metal RCCE library with BareMichael "out of the box." While some features such as dynamic memory management are properly implemented for general use, others, including virtual memory management functions and file operations, are tailored to be compatible with RCCE, though not fully implemented to fulfil their intended duties. These functions are not necessarily safe for use outside of the purpose of supporting RCCE V2.0.

We now present some performance results for RCCE V2.0 running in the BareMichael environment. The simple "ping-pong" benchmark [5] was run on cores 0 and 1 with the SCC mesh and memory running at 800 MHz and core clocks of 533 MHz. As seen in Figure 3, the benchmark exhibits nearly identical performance regardless of whether it is run within

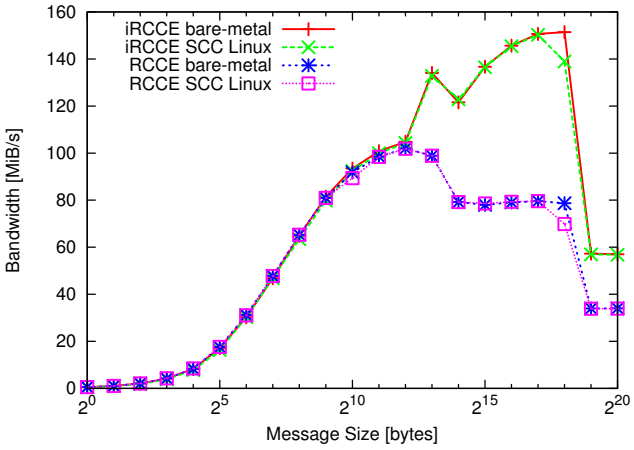


Fig. 3. Ping-pong benchmark results for RCCE and iRCCE running in SCC Linux and BareMichael environments.

SCC Linux or BareMichael. The same is seen when running the benchmark with the pipelining send and receive functions of iRCCE V1.2 [6] in both environments.

IV. Implementation of Xipx OS

Due to its minimalistic nature, BareMichael is a suitable foundation not only for running individual parallel applications, but also for launching operating system kernels. We demonstrate this with *Xipx*, an SCC port of the Embedded Xinu operating system that leverages the BareMichael framework for hardware initialization [7]. The following section presents the Xipx MPB device, which stands as an asynchronous alternative to the RCCE/iRCCE way of managing the SCC's message passing hardware.

A. The Xipx MPB Device

Xipx exposes the SCC message passing buffers via the standard Xinu device API [8]. Several instances of an MPB device are created at boot time, and each one acts as a two-way message passing channel. As an asynchronous and interrupt-driven driver, the Xipx MPB device facilitates inter-core communications in a way that is fundamentally different than RCCE. The basic RCCE API uses a symmetric name space model, meaning all cores access shared variables in the same way – using a variable name and the core ID of the MPB where the variable is stored. In order to preserve this symmetry, certain RCCE routines must be encountered jointly by all cores involved in the system. These routines are referred to as "collective operations," and saying they are "encountered jointly" means that they get called in the same order with respect to each other on all cores in the system. For example, the `RCCE_malloc(size)` routine, which allocates `size` bytes in the local MPB, is a collective operation – any core calling `RCCE_malloc(size)` is counting on all other cores to do the same in the same order with respect to other collective operations. This ensures that all cores are returned a pointer with the same offset from the beginning of their

 ISBN: 978-2-7257-0016-8

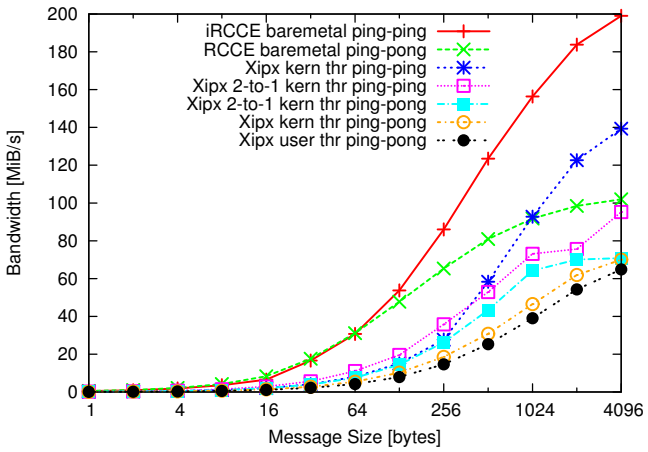


Fig. 4. Benchmark performance of the asynchronous Xipx MPB device.

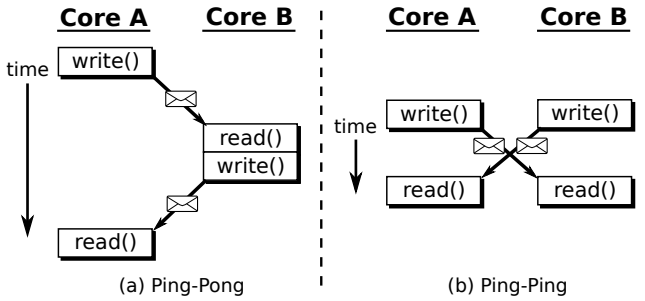


Fig. 5. Comparison of the communication patterns for (a) the ping-pong benchmark and (b) the ping-ping benchmark.

respective MPBs, and they can therefore safely assume the correct location of the corresponding variable in any other remote MPB.

The symmetric name space of a RCCE application is a convenient and efficient way to manage MPB space for a single application, but it is not capable of supporting multiple simultaneous applications on the SCC. With multiple parallel applications running at a time, one cannot guarantee the order in which the applications will get CPU time on each core, and therefore cannot guarantee that collective operations are encountered jointly by all cores. As a simple example, consider two different applications running on the SCC. One of them runs on cores 0 and 1, the other on cores 0 and 2. Because core 0 is involved in both applications, but cores 1 and 2 are only involved in one each, any collective operation core 0 performs in one application is not performed by its communicating partner in the other, therefore the name space symmetry is broken.

In order to support an arbitrary graph of communicating threads on SCC cores, the Xipx MPB device does not assume a symmetric name space. Instead, Xipx treats each MPB as a FIFO buffer. Messages are written to the receiving core's MPB with a header to indicate the core and channel from which it was sent, the channel to which it should be delivered, and the length of the payload. These messages can arrive from any core in any order, and the presence of a new message is signalled by an interrupt. The handler for this interrupt searches through the local MPB devices to find one that is open on the channel indicated by the message header. It then copies the message to a pre-allocated buffer and sends a signal to the thread that owns the device so that a subsequent (or pending) call to `read()` will retrieve the data.

Figure 4 illustrates the performance of the Xipx MPB device for a number of scenarios. All benchmarks were run with the same hardware configuration as described in Section III. In addition to the basic ping-pong benchmark, we executed the "ping-*ping*" benchmark in which two cores each simultaneously send a message to each other and then simultaneously retrieve the message they were sent. The communication patterns of the ping-pong and ping-ping benchmarks are illustrated in Figure 5. In a two-to-one ping-X test, one core runs two simultaneous ping-X benchmarks, each with a different partner core. The measured bandwidth is total data flow in and out of the shared core. For comparison, the RCCE ping-pong performance is duplicated here and iRCCE ping-ping data is introduced.

The Xipx MPB device achieves about 70% of the bandwidth of RCCE and iRCCE in ping-pong and ping-ping benchmarks, respectively. Xipx kernel threads slightly outperform user threads due to the overhead associated with user thread system calls. Though the Xipx device does not match the two libraries in raw bandwidth, the two-to-one benchmarks it performs are not even possible with those libraries. As we have already discussed, this is because the collective communications on which the libraries rely prohibit their use in two concurrent programs. Furthermore, the absence of a RCCE ping-ping benchmark is due to the fact that the library's synchronous semantics render it incapable of implementing that communication pattern.

B. *Porting an OS with BareMichael*

Xipx is, in fact, the precursor of BareMichael; the framework was extracted from Xipx as the initial set of operations that set up a C execution environment in 32-bit protected mode. Due to this development history, the authors cannot comment on the effort required to port another x86-based OS to the SCC using BareMichael as an aide. However, Xipx diverges from BareMichael beginning in the `startup()` routine, and we believe that the execution path preceding that point is generic enough to be useful for other operating systems as hardware initialization code. The `initPaging()` routine may be replaced or modified to set up an appropriate initial pagetable. Furthermore, regardless of the build process used to generate the OS image, the build environment of BareMichael should be useful for merging that image with the framework's initialization code and loading the resulting composite image into SCC memory.

V. RELATED WORK

Microsoft has released a Visual Studio add-in and bare-metal environment package for the SCC [9]. The source

code for the minimalistic bare-metal environment is provided, thereby allowing developers to modify the environment to suit their needs. However, development options are limited as the Microsoft tools must be run from a Windows machine that has a network connection to the MCPC. Furthermore, the license for this framework allows for non-commercial use only, and it grants back to Microsoft the right to use, modify, and sell any modifications to and/or derivative works of their framework. BareMichael tools are run directly on the MCPC, and its open-source, BSD-style license is less restrictive, permitting redistribution and use of the framework and derivative works, both in source and binary forms, for both commercial and non-commercial purposes.

ETI provides a beta version of its SCC Development Framework [10] for compiling and launching bare-metal applications on the SCC. Applications get compiled into an ELF format binary and are loaded and launched via a utility running on the MCPC. The ETI framework is closed source, and therefore lacks certain flexibilities offered by BareMichael such as the ability to modify the boot process. The current release also offers no means of specifying which cores to boot up, only the number of cores. Furthermore, it forces the same image to be loaded onto all cores at once. In contrast, BareMichael is able to load different images to different cores with only minor changes to the build process.

Finally, an Intel internal framework named *BareMetalC* exists [5], but it is not released to the public due to licensing limitations. The bare-metal RCCE library was created specifically to support this framework.

VI. CONCLUSION

We have introduced BareMichael, a minimalistic, open-source framework for loading and executing bare-metal programs on the Intel SCC architecture. Our lightweight framework is packaged with a subset of the standard C library, and features out-of-the-box support for Intel's message-passing library, RCCE. A basic benchmark shows that message-passing bandwidth for RCCE on bare-metal is nearly identical to that for RCCE in SCC Linux.

A programmer developing in BareMichael is not limited merely to launching individual parallel applications on the SCC. The flexibility of the framework is demonstrated by our implementation of Xipx, a port of the Embedded Xinu operating system, for which BareMichael serves as the foundation. In order to allow multiple threads to simultaneously use the SCC's message passing buffer in a preemptive environment, Xipx manages the MPB hardware at the device layer. Though our simple device implementation does not match the bandwidth of RCCE, it allows for asynchronous communications and allows multiple processes to use the MPB simultaneously, two features that are not possible with the basic RCCE API. Future work on Xipx will investigate how to increase message-passing performance at the device layer.

Typical usage of the SCC involves loading the private memory of each core with an identical image. However, there is interest in being able to boot different images on different cores [11], [12]. The utilities of sccKit allow for this, and we have been successful in using the BareMichael build environment to load and boot heterogeneous images on the SCC. A future release will incorporate this functionality.

The current, simplistic implementation of MikeTerm only allows one-way serial communication from the SCC to the MCPC. Two-way communication is desirable, and it may be realized via the UART support that was introduced with version 1.4.2 of sccKit. We plan to look into this possibility for future releases as well.

We provide the BareMichael framework as an open-source package in the hope that it will lower the entry barrier for others wishing to develop and run bare-metal applications on the Intel SCC. The framework is available for download at http://marcbug.scc-dc.com/svn/repository/trunk/baremetal/baremichael/.

ACKNOWLEDGMENT

The authors would like to thank the members of the Intel Many-core Applications Research Community – in particular Ted Kubaska and Jan-Arne Sobania – for their prompt and clear responses to questions arising during development on the SCC. Thanks also to Intel Corporation for access to the SCC hardware.

REFERENCES

[1] J. Howard, S. Dighe, S. R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. R. M. Gries, G.Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. K. De, and R. V. der Wijngaart, "A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 173–183, Jan. 2011.

[2] *SCC External Architecture Specification (EAS)*, Intel Corporation, Nov. 2010, revision 1.1.

[3] D. Brylow and B. Ramamurthy, "Nexos: A next generation embedded systems laboratory," *SIGBED Review*, vol. 6, no. 1, Jan. 2009, URL http://sigbed.seas.upenn.edu/.

[4] T. Mattson and R. van der Wijngaart, "RCCE: A small library for many-core communication," Intel Corporation, Jan. 2011, software Version 2.0-release.

[5] 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 *Proceedings of the 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. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.53

[6] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and improvements of programming models for the Intel SCC many-core processor," in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, Jul. 2011, pp. 525–532.

[7] M. W. Ziwisky, "A message-passing, thread-migrating operating system for a non-cache-coherent many-core architecture," Master's thesis, Marquette University, to be published.

[8] D. E. Comer, *Operating System Design: The XINU Approach, Linksys Version*. CRC Press, 2011.

[9] (2011, Mar.) Visual Studio add-in and bare-metal environment for Intel SCC. Microsoft Research. [Online]. Available: http://research.microsoft.com/en-us/downloads/37ccb116-c67d-4c44-9181-898889b8352d/

[10] (2011, Aug.) ETI's SCC development framework available. Intel MARC forums. [Online]. Available: http://communities.intel.com/thread/17643/

[11] (2011, Jun.) Booting custom kernels on the SCC. Intel MARC forums. [Online]. Available: http://communities.intel.com/message/128484/

[12] (2011, Sep.) rccerun to start different applications on cpu cores. Intel MARC forums. [Online]. Available: http://communities.intel.com/message/137635/