# A Runtime Implementation of Go's Concurrency Constructs on the Single-Chip Cloud Computer

Andreas Prell
andreas.prell@uni-bayreuth.de

October 17, 2011

## 1 Introduction

Go is a general-purpose programming language intended for systems programming [1]. We don't provide a general description of Go, but instead we 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 "not communicate by sharing memory" but "to share memory by communicating." This style of programming is reminiscent of message passing, where messages are used to exchange data between concurrently executing processes and to coordinate execution. 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 can be found elsewhere: in programming Intel's Single-Chip Cloud Computer (SCC) research processor. The SCC is intended to foster manycore software research, on a platform that's more like a "cluster-on-a-chip" than a traditional shared-memory chip multiprocessor. As such, the SCC is tuned for message passing rather than for "threads and locks". Or as 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 ask the question, isn't Go's concurrency model a perfect fit for such a processor architecture? To find out, we start by implementing the necessary runtime support on the SCC.

## 2 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. What all these languages have in common is 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). 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 turn in the following sections.

### 2.1 Goroutines

Think of goroutines as lightweight threads that run concurrently with other goroutines, including the calling code. Whether a goroutine runs in a separate thread or whether multiple goroutines are multiplexed onto the same thread 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." [5] In other words, a `go` statement marks an asynchronous function call that doesn't wait until the goroutine returns before continuing with the next statement that follows after it.

### 2.2 Channels

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." [5] 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.

## 3 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 many-core architectures and the techniques used to program them. The principal idea behind the SCC is to abandon cache coherence with all the associated protocol overhead and to adopt a programming model that has proven to be extremely scalable—message passing. The result looks and is programmed very much like a cluster of workstations, integrated on a single piece of silicon.

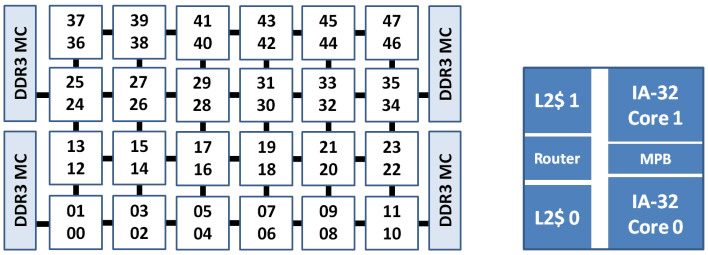


Figure 1: The SCC processor: 6x4 tile array (left), 2-core tile (right)

## 3.1 Hardware Overview

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, making the SCC appear as a "cluster-on-a-chip".

Figure 1 gives an idea of what the SCC looks like. It's a tiled multicore processor with 24 tiles arranged in a 6x4 mesh on-die network. Each tile contains two Pentium-class IA-32 cores (P54C), each with 16 KB L1 instruction and data caches, 256 KB L2 cache per core, 16 KB shared SRAM, called the Message Passing Buffer (MPB), and a router for inter-tile communication. The name of the shared SRAM suggests why it has been included in the first place, namely as a communication buffer to support low-latency message passing between cores.

## 3.2 Native Programming Model

One of the goals with the SCC is to explore scalable programming models, so it comes as no big surprise that the native programming model is message passing, using an API, known as RCCE (pronounced "rocky"), that should look familiar to users of MPI [6]. Listing 1 shows an example RCCE program that uses message-passing to shift all worker IDs to the right by one. For convenience, RCCE programs are usually run with the command `rccerun`, which is basically a wrapper script around `pssh` to load an executable on a specified number of processors (cores). In addition, `rccerun` makes sure that the MPBs and test-and-set registers are cleared before starting the program. So, assuming program 1 is run on three cores numbered 0, 1, and 2, core 0 will print 2, core 1 will print 0, and core 2 will print 1, in some undetermined order.

The program we show here uses RCCE's high-level interface, which provides send and receive routines without exposing the underlying communication. RCCE also has 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 includes an API to vary voltage and frequency within domains of the SCC, but we won't go into power management issues here.

```
1  int main(int argc, char *argv[])
2  {
3    int ID, NUES, next, prev;
4    int buf, i;
5
6    RCCE_init(&argc, &argv);
7
8    ID = RCCE_ue();
9    NUES = RCCE_num_ues();
10   next = (ID + 1) % NUES;
11   prev = (ID - 1 + NUES) % NUES;
12
13   for (i = 0; i < 2; i++) {
14     if ((ID + i) % 2 == 0)
15       RCCE_send(&ID, sizeof(ID), next);
16     else
17       RCCE_recv(&buf, sizeof(buf), prev);
18   }
19
20   printf("Worker %d: %d\n", ID, buf);
21
22   RCCE_finalize();
23
24   return 0;
25 }
```

Listing 1: A simple RCCE program that exchanges messages between threads running on different cores (units of execution or UEs in RCCE jargon).

# 4 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 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. Obviously, if we want to go beyond MPI-like message passing, 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. 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. What worker `j` then needs to know is `b`'s offset within `i`'s MPB. This offset `o` is easily determined by

```
int o = (int)b - (int)RCCE_comm_buffer[i];
```

and after communicating `o` to worker `j`, `j` can get a pointer to `b` through

```
void *b = (void *)((char *)RCCE_comm_buffer[i] + o);
```

and use this pointer to access whatever is stored at this address. To summarize, passing around (ID, offset) tuples allows us to break with the collective allocations model of RCCE and use the MPBs more like local stores.

## 4.1 Goroutines as Tasks

We have previously implemented a tasking environment to support dynamic task parallelism on the SCC [7]. 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 later 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, is designated to run the main program between the initializing and finalizing calls to the tasking environment. This thread can naturally call goroutines, but it cannot itself schedule goroutines for execution. We need to have at least one worker thread to be able to run goroutines. This is more or less a restriction imposed by the tasking environment, but we live with that for now.

The `go` statement and its translation into C using our runtime library is currently being implemented as a language extension for the xoc compiler [8]. Xoc is a C compiler frontend (a source-to-source translator) whose focus on extensibility makes it comparatively easy to prototype new language constructs.

Figure 2 shows a pictorial representation of workers running goroutines (tasks). 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 create a coroutine for every goroutine they schedule to be able to transfer control from one goroutine to another. While a goroutine shares the address space with other goroutines created by the same worker, goroutines created by different workers also run in different address spaces (sharing memory is possible). This is a deviation from the 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 to communicate, regardless on which core they are running. Channels in shared memory provide such a mechanism.

## 4.2 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 has to be lock-based because the SCC lacks atomic operations and only provides a small number of test-and-set registers (one 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 another item is send to the channel, the sender blocks until an item

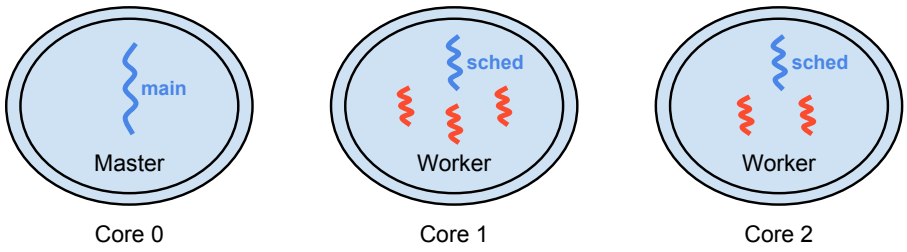


Figure 2: An example execution of a program with goroutines on three cores of the SCC processor. In this example, worker 1 is running three goroutines, while worker 2 is running two goroutines. The master thread can call goroutines but not run them.

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 the receive has happened.

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).

## 4.3 Channel API

**Basic channel functions**

`Channel *channel_alloc(size_t size, size_t n);`
Allocates a channel for elements of `size` 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 `size` bytes. Also note that the current implementation does not allow all combinations of `size` and `n`. This is because the underlying allocator works with cache line granularity, so we have to make sure that channel buffers occupy multiples of 32 bytes (`(n+1)*size` must be a multiple of 32).

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

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

`bool channel_receive(Channel *chan, void *data, size_t size);`
Receives an element of `size` bytes from channel `chan`. The element is stored at ad-

dress `data`. The call blocks if the channel is empty.

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

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

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

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

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

## 4.4 Toy Examples

**Fibonacci** The main function in Listing 2 spawns a goroutine to print the Fibonacci series up to the $n$th number. We use an unbuffered channel to signal completion after the goroutine has done its work. The thread running main between the initializing and finalizing calls to `TASKING_init` and `TASKING_exit` (the master thread) cannot schedule goroutines for execution, so it will block on the channel in line 50 until `print_numbers` has signaled completion in line 35. Thus, the program requires at least one more worker thread to run correctly. Worker threads pick up goroutines for execution and switch between goroutines that are blocked on channels. Function `print_numbers` spawns `produce_numbers` as another goroutine with which it communicates through two channels. The parameter `n` is exchanged over an unbuffered channel `chan1`. The $n + 1$ Fibonacci numbers that are computed by `produce_numbers` are exchanged over a buffered channel `chan2` with a capacity to hold up to seven numbers. When we run the program on two cores (master + one worker), the worker runs both goroutines and switches between them as needed, always sending and receiving the next seven numbers. When we run the program on three cores (master + two workers), the two goroutines are placed on different cores and run in parallel.

**Prime Sieve** The prime sieve example in Listing 3 closely follows the Go code presented in the Go language specification [5] and in the Go tutorial [9]. Numbers are passed through a dynamic chain of goroutines. Each goroutine receives an input stream of numbers, prints the first number in the stream (a prime), and passes all numbers that are not multiples of the prime from the input to the output. Because we leak MPB

memory in line 36, we can print only a few dozen prime numbers before running out of MPB memory. An unbuffered channel takes up at least 160 bytes (four cache lines to hold the data structure, plus a cache line of internal channel buffer). Go, on the other hand, has a garbage collector, which reclaims memory behind the scenes, and which, according to Rob Pike, is "essential to easy concurrent programming". [10] We could support a much larger number of channels in shared off-chip memory if communication latency is of secondary importance, but that doesn't help with the memory leak in this program.

# 5 Conclusion

We have presented a runtime implementation of Go's concurrency constructs (goroutines and channels) on a novel processor architecture, Intel's SCC manycore research chip. Both Go and the SCC share the basic idea of communicating and synchronizing over messages rather than shared memory. Channels can be implemented very efficiently using the available hardware support for low-latency messaging. One problem in practice might be the small size of the on-chip memory, which limits the number of channels that can be used simultaneously, as well as the size and number of data items that are exchanged. Channels are certainly an interesting mechanism for coordinating concurrently executing activities, regardless whether these are goroutines or tasks in general. Building on our implementation, we plan to look into the design of message-passing schedulers and see if channels can be used to handle all communication in such systems.

# References


[1] The Go Programming Language. `http://golang.org`.

[2] Many-core Applications Research Community. `http://communities.intel.com/message/113676#113676`.

[3] Many-core Applications Research Community. `http://communities.intel.com/message/115657#115657`.

[4] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21:666–677, August 1978.

[5] The Go Programming Language Specification. `http://golang.org/doc/go_spec.html`.

[6] Timothy G. Mattson et al. 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*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

```
1  void produce_numbers(Channel *chan1, Channel *chan2)
2  {
3    int fib = 0, f1 = 1, f2 = 0, i, n;
4
5    channel_receive(chan1, &n, sizeof(n));
6
7    for (i = 0; i <= n; i++) {
8      if (i < 2) {
9        channel_send(chan2, &i, sizeof(i));
10       continue;
11     }
12     fib = f1 + f2;
13     f2 = f1;
14     f1 = fib;
15     channel_send(chan2, &fib, sizeof(fib));
16   }
17 }
18
19 void print_numbers(Channel *chan, int n)
20 {
21   Channel *chan1, *chan2;
22   int fib, done, i;
23
24   chan1 = channel_alloc(32, 0);
25   chan2 = channel_alloc(4, 7);
26   go produce_numbers(chan1, chan2);
27   channel_send(chan1, &n, sizeof(n));
28
29   for (i = 0; i <= n; i++) {
30     channel_receive(chan2, &fib, sizeof(fib));
31     printf("fib(%d) = %d\n", i, fib);
32   }
33
34   done = 1;
35   channel_send(chan, &done, sizeof(done));
36   channel_free(chan1);
37   channel_free(chan2);
38 }
39
40 int main(int argc, char *argv[])
41 {
42   Channel *chan;
43   int n = 42, done = 0;
44
45   RCCE_init(&argc, &argv);
46   TASKING_init();
47
48   chan = channel_alloc(32, 0);
49   go print_numbers(chan, n);
50   channel_receive(chan, &done, sizeof(done));
51   assert(done == 1);
52   channel_free(chan);
53
54   TASKING_exit();
55   RCCE_finalize();
56
57   return 0;
58 }
```

Listing 2: A toy example that prints the Fibonacci series up to the $n$th number.

```
1  // Send the sequence 2, 3, 4, ... to channel chan
2  void generate(Channel *chan)
3  {
4    int i;
5
6    for (i = 2; ; i++)
7      channel_send(chan, &i, sizeof(i));
8  }
9
10 // Copy the values from channel in to channel out,
11 // removing those divisible by prime
12 void filter(Channel *in, Channel *out, int prime)
13 {
14   int i;
15
16   for (;;) {
17     channel_receive(in, &i, sizeof(i));
18     if (i % prime != 0)
19       channel_send(out, &i, sizeof(i));
20   }
21 }
22
23 // Print the first n prime numbers
24 void sieve(int n)
25 {
26   Channel *chan;
27   int prime, i;
28
29   chan = channel_alloc(32, 0);
30   go generate(chan);
31   for (i = 0; i < n; i++) {
32     Channel *chan1 = channel_alloc(32, 0);
33     channel_receive(chan, &prime, sizeof(prime));
34     printf("%d\n", prime);
35     go filter(chan, chan1, prime);
36     chan = chan1;
37   }
38 }
39
40 int main(int argc, char *argv[])
41 {
42   RCCE_init(&argc, &argv);
43   TASKING_init();
44
45   sieve(30);
46
47   TASKING_exit();
48   RCCE_finalize();
49
50   return 0;
51 }
```

Listing 3: A toy example that prints the first $n$ prime numbers. This program appears in the Go language specification [5] and in the Go tutorial [9].

[7] Andreas Prell and Thomas Rauber. Task Parallelism on the SCC. In *Proceedings of the 3rd Many-core Applications Research Community (MARC) Symposium*, MARC 3, pages 65–67. KIT Scientific Publishing, 2011.

[8] Russ Cox, Tom Bergan, Austin T. Clements, Frans Kaashoek, and Eddie Kohler. Xoc, an Extension-Oriented Compiler for Systems Programming. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 244–254, New York, NY, USA, 2008. ACM.

[9] A Tutorial for the Go Programming Language. `http://golang.org/doc/go_tutorial.html`.

[10] Go Emerging Languages Conference Talk. `http://www.oscon.com/oscon2010/public/schedule/detail/15299`, July 2010.