Automatic Buffering of Rendezvous Models

Ravindra Babu Ganapathi Nalini Vasudevan Stephen A. Edwards
Columbia University, New York
rg2547@columbia.edu, naliniv@cs.columbia.edu, sedwards@cs.columbia.edu

Abstract

Many concurrent programming models use rendezvous communication: the sender and the receiver both waiting for each other for the communication to succeed. There are two problems with this model. Firstly, the sender cannot go ahead to do some other computation. Therefore, there is a performance bottleneck. Secondly, a program with these blocking constructs may be susceptible to deadlocks.

A remedy to these problems is to use bounded buffers instead of rendezvous communication. In this paper, we implement the rendezvous communication of SHIM, a concurrent programming language with bounded buffers and discuss methods to automate it. SHIM is a deterministic language: its input-output behavior is not affected by the scheduling choices taken by the operating system. By increasing the number of places in the buffer, we still maintain the determinism and the characteristics of the original program, except that we resolve deadlocks in certain cases.

1. Motivation

SHIM [Edwards and Tardieu 2006] is a C-like concurrent programming language that combines Kahn's[Kahn 1974] determinism with Hoare's[Hoare 1978] rendezvous communication. Figure 1 is a SHIM program in which two tasks *foo* and *bar* run concurrently and communicate on channels *c* and *d*. The first *send* in *foo* communicates the value 5 on channel *c* and waits for *bar* to receive the value. The tasks rendezvous on channel *c*, then continue to the next statement. The two tasks then rendezvous on *c* again to exchange the value 10. Then the two tasks rendezvous on *d* followed by the rendezvous on *c*.

Since the sender is blocking, the the first call to *baz* cannot execute until the thread executing *bar* finishes the first two receives on *c*. Instead, if we use buffers for *c*, then the sends on *c* in foo does not have to wait to rendezvous and can go ahead with the computation of *baz*. In this example *c* requires a 2-place buffer to guarantee that the two sends in *foo* never block, irrespective of the state of the other thread executing *bar*. *c* does not require more than 2 places because the *recv d* in *baz* has to wait for a matching *send d*. The *recv d* therefore succeeds only after the first two *recv c*'s and the *send d* in *bar* succeeds. This forces $c's$ buffer to be empty during *recv d*.

SHIM supports multi-way rendezvous - multiple receivers may rendzvous on a *send*. Figure 2(b) is the buffered equivalent of the

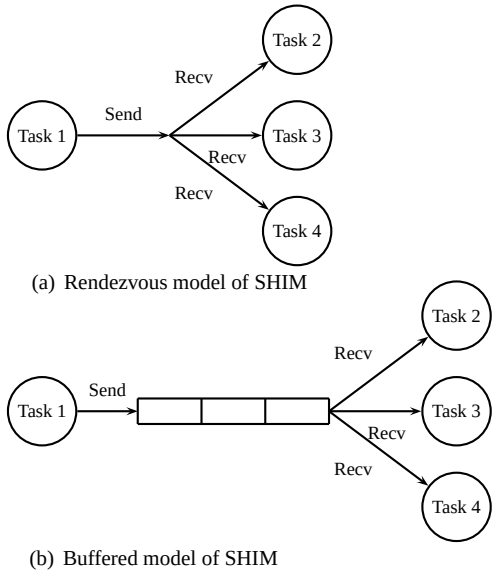
[Copyright notice will appear here once 'preprint' option is removed.]

```
void foo(chan out c, chan in d)
  int y = 9;
  send c = 5;
  send c = 10;
  baz(y);
  y += recv d;
  send c = 15;
  baz(y);
}

void bar(chan in c, chan out d) {
   int z = 9;
   qux(z);
   z += recv c;
   z += recv c;
   send d = z;
   z += recv c;
   qux(z);
}

main() {
  chan int c, d;
  foo(c, d) par bar(c, d);
}
```

Figure 1. A program in which two tasks rendezvous on channel *c*



(a) Rendezvous model of SHIM

(b) Buffered model of SHIM

Figure 2. From rendezvous communication to buffered communication

rendezvous model of Figure 2(a). We found that moving from 0 place-buffer (rendezvous) to one or more places gives increased performance for many of our examples in SHIM. See Figure 3 and Figure 4. The number of places denotes the bound on the buffer.

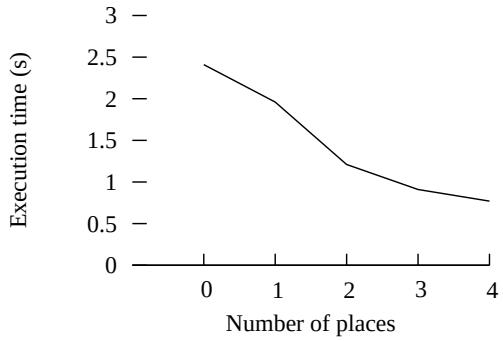


Figure 3. Times for Pipeline

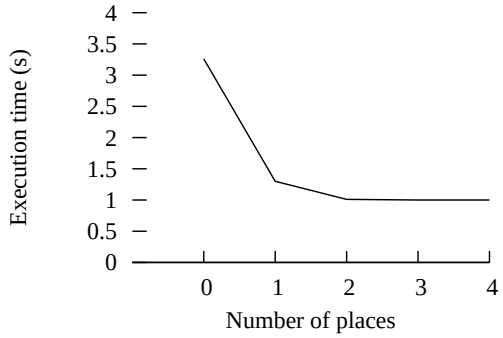


Figure 4. Times for JPEG Decoder

2. Approach

In this section, present a static analysis method that determines the upper bound of the number of places required in a buffer.

A SHIM task alternates between computation and communication. As a first step, we do not make any assumption about the architecture and hence the computation time. We therefore abstract away the computation and then find the number of places required independent of the computation time. We also abstract away the details of data manipulation and assume all branches of any conditional statement can always be taken at any time.

Channel c of the program in Figure 1 needs a 2-place buffer to guarantee that the *send*s in foo does not block. d requires only a 1-place buffer. To find the bound automatically, we convert the program into a petri-net[Murata 1989] and determine the maximum number of tokens that can accumulate in the place representing the channel's buffer. Figure 5 is the petri-net representation of the progam in Figure 1. $p1$ and $p2$ at the center represent the buffers for c and d. By simulating the petri-net, we can determine the maximum number of tokens that can accumulate at these locations, which gives the upper bound on the number of places required for channels c and d. On simulating this petri-net, the maximum number of tokens that can accumulate in $p1$ is 2 and $p2$ is 1.

We currently rely upon the assumption that the channel connections can be determined statically. Also, our algorithm may return an unrealistic bound like infinity. In such cases, we divide the available memory equally among the channels.

We seek to improve our technique by using statically profiled data to determine the computation overhead for a given problem and hence identify the number of places required for a channel. Finally, we will calculate the optimal number of places with a combination of both our techniques.

Currently, we improve the performance of our tool by giving priorities to transitions - for example, allow the sender to go faster than the receiver since the receiver has to anyway wait. We also plan to look at compositional techniques that will scale our tool better.

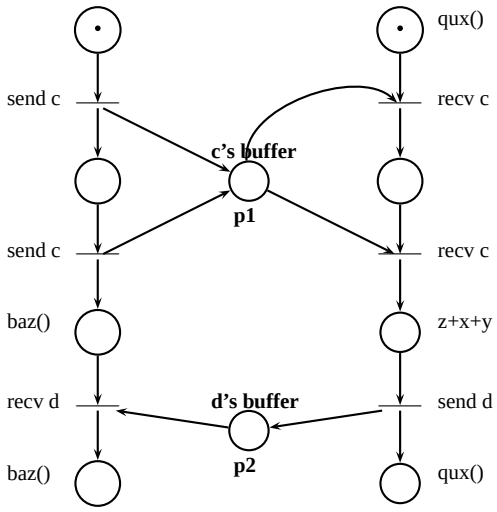


Figure 5. Petri-net representation of the program in Figure 1

3. Conclusions

We have presented an automated method for buffering SHIM, an enhancement over the rendezvous SHIM retaining its semantics and deterministic behavior.

In certain cases, we found that the buffered communication resolves deadlocks caused by cyclic waiting of the communicating processes. Our next plan is to automatically resolve deadlocks by using an appropriate buffer bound.

We currently focus our techniques on SHIM. We would like to apply them on other rendezvous languages like Ada and MPI. Finally, our research objective is to make concurrent programming easier by providing simple, fast and a deterministic behavior. We plan to achieve this by hiding the complexity in the compiler.

References

Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):854–867, August 2006. URL `http://dx.doi.org/10.1109/TVLSI.2006.878473`.

C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland.

Tadao Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.