Submitted to MEMOCODE '09

Buffer Sharing in CSP-like Programs

Nalini Vasudevan
Columbia University, New York
naliniv@cs.columbia.edu

Stephen A. Edwards
Columbia University, New York
sedwards@cs.columbia.edu

Abstract

Most compilers focus on optimizing performance, often at the expense of memory, but efficient memory use can be just as important in constrained environments such as embedded systems.

In this paper, we present a memory reduction technique for the deterministic concurrent programming language SHIM. We focus on reducing memory consumption by sharing buffers among the tasks, which use them to communicate using CSP-style rendezvous. We determine pairs of buffers that can never be in use simultaneously and use a shared region of memory for each pair.

Our technique produces a static abstraction of a SHIM program's dynamic behavior, which we then analyze to find buffers that can share memory. Experimentally, we find our technique runs quickly on modest-sized programs and often reduces memory requirements by half.

Keywords Concurrency, SHIM, Static Analysis, Buffers, Optimization

1. Introduction

Embedded systems have limited memory. Overlays, which amount to time-multiplexing the use of memory regions, is one way to reduce a program's memory consumption. In this paper, we propose a technique that automatically finds opportunities to safely overlay communication buffer memory in a concurrent programming language.

The technique we present here determines what buffer memory may be shared in SHIM programs (Edwards and Tardieu 2005; Tardieu and Edwards 2006a,b). This is closely related to some of the techniques used by Vasudevan and Edwards (2008), although we solve a different problem.

SHIM is an asynchronous concurrent language and is scheduling-independent: its input/output behavior is not affected by any nondeterministic scheduling choices taken by its runtime environment, due to processor speed, the operating system, scheduling policy, etc. A SHIM program is composed of sequential tasks that synchronize whenever they want to communicate. The language is a subset of Kahn networks (Kahn 1974) (to ensure determinism) that employs the rendezvous of Hoare's CSP (Hoare 1985) for communication to keep its behavior tractable.

SHIM processes communicate through channels. The sequence of symbols transmitted over each channel is deterministic, although the relative order of symbols between channels is generally undefined. If the sequences of symbols transmitted over two channels do not interfere, then we can safely share buffers. We propose a technique for establishing ordering between pairs of channels; if such ordering cannot be established, we conclude that the pair cannot use the same buffers.

Our analysis is conservative: if we establish two channels can share buffers, they can do so safely, but we may miss opportunities to share certain buffers because we do not model data and may treat the program as separate pieces to avoid an exponential explosion in analysis cost. Specifically, we build sound abstractions to avoid state space explosions, effectively enumerating all possible schedules with a product machine.

One application of our technique is to minimize buffer memory used by code generated by the SHIM compiler for the Cell Broadband engine (Vasudevan and Edwards 2009). The heterogeneous Cell processor (Kahle et al. 2005) consists of a power processor element (PPE) and eight synergistic processor elements (SPEs). The SHIM compiler maps tasks onto each of the SPEs. Each SPE has its own local memory and shares data through the PPE. The PPE synchronizes communication and holds all the channel buffers in its local memory. The SPE communicates with the PPE using mailboxes (Kistler et al. 2006).

We wish to reduce memory used by the PPE by overlapping buffers of different channels. Our static analyzer does liveness analysis on the communication channels and determines pairs of buffers that are never live at the same time. We demonstrate in Section 7 that the PPE's memory usage can be reduced drastically for practical examples such as a JPEG decoder and an FFT.

Below, we describe the SHIM language (Section 2), how we model its behavior to analyze buffer usage (Section 3), how we compose models of SHIM tasks to build a product machine for the whole program (Section 4), how we avoid state explosion while doing this (Section 5), and how we use these results to reduce buffer memory usage (Section 6). We present experimental results in Section 7 and related work in Section 8.

2. The SHIM programming language

SHIM (Edwards and Tardieu 2005; Tardieu and Edwards 2006a,b) is a C-like concurrent programming language. Tasks in SHIM communicate through multi-way rendezvous channels. To the usual collection of C-like expressions and statements it adds two constructs: *par* for specifying concurrency and *next* for communication. *p par q* runs statements *p* and *q* in parallel and finishes when both *p* and *q* terminate. *next c* is the communication construct that synchronizes on channel *c*. It sends data if it appears on the left side of an assignment and receives data otherwise. To preserve determinism, SHIM has no global or shared variables.

In Figure 1, two tasks run concurrently within *main* and communicate on channels *a* and *b*. The *next a* in task 1 is a send because it appears on the left side of the assignment. The *next a* of task 2 is a receive. Similarly, the *next b* of task 2 is a send and *next b* of task 1 is a receive. The *next a* in task 1 assigns 6 to *a* and waits for task 2 to receive the value. The tasks therefore rendezvous at their *next*s, then continue to the next statement. Next, the two tasks rendezvous at *next b*. There, task 1 receives the value 8 from task 2.

If there are two or more senders on a particular channel, the compiler simply rejects the program. If the statements *next a* and *next b = 8* were interchanged, the program would deadlock.

SHIM compiles to C. Back ends produce code for a variety of environments: shared-memory multiprocessors using the pthreads library (Edwards et al. 2008), the IBM Cell Broadband Engine (Va-

```
void main()
{
 chan int a, b;
 { // Task 1

    next a = 6; // Send 6 on a (synchronize with task 2)
    // a = 6 here
    next b; // Receive b (synchronize with task 2)
    // b = 8 here

 } par { // Task 2

    next a; // Receive a (synchronize with task 1)
    // a = 6 here
    next b = 8; // Send 8 on b (synchronize with task 1)
    // b = 8 here

   }
}
```

Figure 1. A SHIM program in which two tasks exchange data on channels *a* and *b*

```
void main()
{
 chan int a, b, c;
 { // Task 1

    next a = 6; // Send a (synchronize with task 2)

 } par { // Task 2

    next a; // Receive a (synchronize with task 1)
    next b = a + 1; // Send 7 on b (synchronize with task 3)

 } par { // Task 3

    next b; // Receive b (synchronize with task 2)
    next c = b + 1; // Send 8 on c (synchronize with task 4)

 } par { // Task 4

    next c; // Receive c (synchronize with task 3}
    // c = 8 here

 }
}
```

Figure 2. A SHIM program to illustrate the need for buffer sharing

sudevan and Edwards 2009), and single-threaded processors that do not require thread support (Edwards and Tardieu 2006a). SHIM has also been implemented as a library for Haskell (Vasudevan et al. 2008). Hardware translation has also been proposed by Edwards and Tardieu (2006b) but has not yet been implemented.

In this paper we address an optimizing technique for SHIM – buffer sharing. In the program in Figure 2, the main task starts four tasks in parallel. Tasks 1 and 2 communicate on *a*. Then, tasks 2 and 3 communicate on *b* and finally tasks 3 and 4 on *c*. The value of *c* received by task 4 is 8. Communication on *a* cannot occur simultaneously with that of *b* because task 2 sequentializes them. Similarly communications on *b* and *c* are sequentialized by task 3. Communications on *a* and *c* cannot occur together because they are sequentialized by the communication on *b*. Our tool understands this pattern and reports that *a*, *b* and *c* can share buffers because their communications never overlap, thereby reducing the total buffer requirements by 66% for this program.

3. Abstracting SHIM Programs

First, we assume that a SHIM program has no recursion. Edwards and Zeng (2008) show how to remove bounded recursion, which makes the program finite, rendering the buffer minimization problem decidable. We do not attempt to analyze programs with unbounded recursion.

Although the recursion-free subset of SHIM is finite-state and therefore tractable in theory, in practice the state space of even a small program is usually too large to analyze exactly; a sound abstraction is necessary. A SHIM task has both computation and communication, but because buffers are used only when tasks communicate, we abstract away the computation.

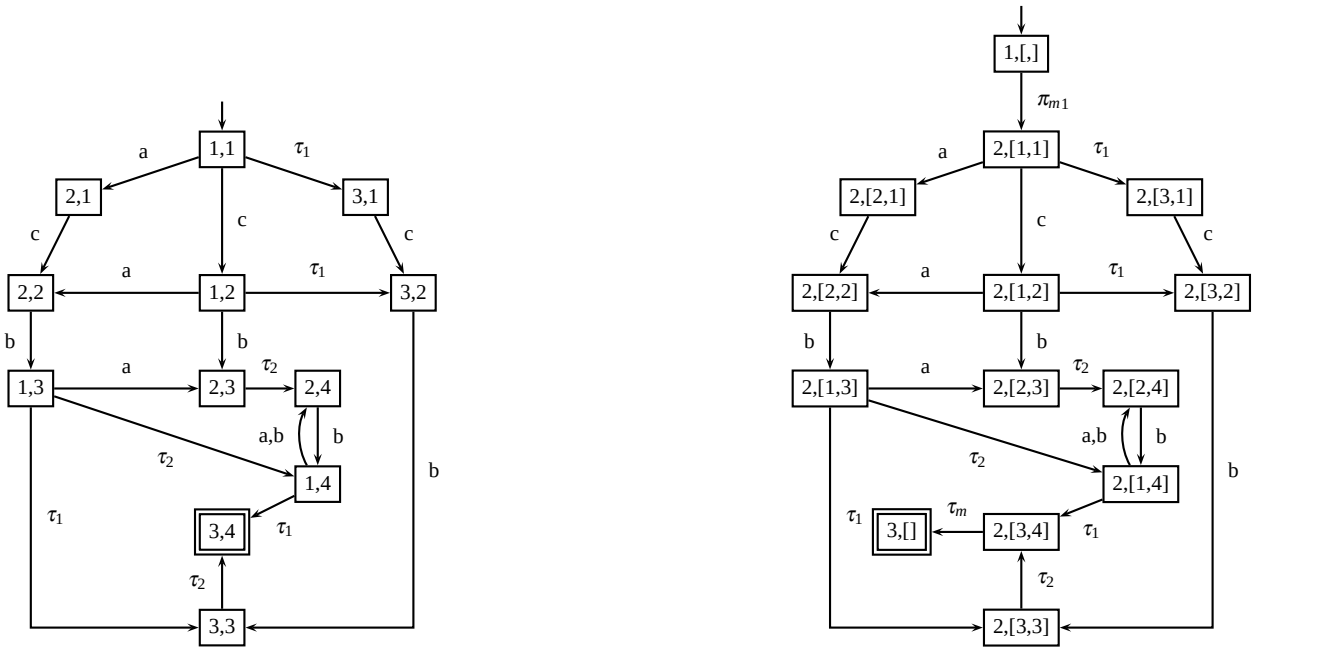
Since we abstract away computation, we must assume that all branches of any conditional statement can be taken. This leaves open the possibility that two channels may appear to be used simultaneously but in fact never are, but we believe our abstraction is reasonable. In particular it is safe: we overlap buffers only when we are sure that two channels can never be used at the same time regardless of the details of the computation.

```
void main() {
  chan int a, b, c;
  { // Task 1

    for (int i = 0; i < 15; i++) { // state 1
        if (i % 2 == 0)
            next a = 5;
        else
            next b = 7;
        // state 2
        next b = 10;
    }
    // state 3

  } par { // Task 2

        // state 1
        next c = 13;
        // state 2
        next b;
        // states 3 & 4
  }
}
```

Figure 3. A (contrived) SHIM program with a loop, conditionals, and a task that terminates

3.1 An Example

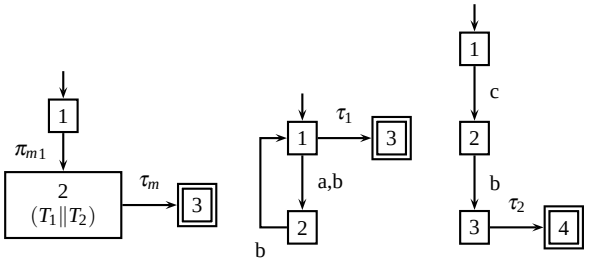
Consider the SHIM program in Figure 3. The *main* function starts two tasks that communicate through channels *a*, *b* and *c*.



(a) $T_1 \cdot T_2$ (a,b,c,τ_1,τ_2)

(b) $M \cdot T_1 \cdot T_2$(a,b,c,τ_1,τ_2,π_{m1})

Figure 5. Composing tasks in Figure 4: (a) Merging T_1 and T_2. (b) Inlining $T_1 \cdot T_2$ in M.



(a) M (a,b,c,π_{m1}) (b) T_1 (a,b,τ_1) (c) T_2 (b,c,τ_2)

Figure 4. The main task and its subtasks

The first task communicates on channels *a* and *b* in a loop; the second task synchronizes on channels *c* and *b*, then terminates. Once a task terminates, it no longer compelled to synchronize on the channels to which it is connected. Thus after the second task terminates, the first task just talks to itself. A process is said to talk to itself when it is the only process that participates in a rendezvous. Terminated processes do not cause other processes to deadlock.

At compilation time, the compiler dismantles the main function of Figure 3 into tasks T_1 and T_2. T_1 is connected to channels *a* and *b* since a and b appear in the code section of T_1. Similarly T_2 is connected to channels *b* and *c*. During the first iteration of the loop in T_1, T_1 talks to itself on *a*; since no other task is connected to *a*. Meanwhile, T_2 talks to itself on *c*. Then the two tasks rendezvous on *b*, communicating the value 10, then T_2 terminates. During subsequent iterations of T_1, T_1 talks to itself on either *b* twice or *a* and *b* once each.

In the program in Figure 3, communication on *b* cannot occur simultaneously with that on c because T_2 sequentializes the two communications and therefore *b* and *c* can share buffers. On the other hand, there is no ordering between channels *a* and *c*; *a* and *c* can rendezvous at the same time and therefore *a* and *c* cannot share buffers. By overlapping the buffers of *b* and *c*, we can save 33% of the total buffer space.

Our analysis performs the same preprocessing as Vasudevan and Edwards (2008). It begins by removing bounded recursion using the technique of Edwards and Zeng (2008). Next, we duplicate functions to force every call site to be unique. This has the potential of producing an exponential blow-up in code side, but we have not observed it in practice.

At this point, the call graph of the program is tree, enabling us statically determine all the tasks and the channels to which each is connected.

Next we disregard all functions that do not affect the communicating behavior of the program. Because we are ignoring data, their behavior cannot affect whether we consider a buffer to be sharable. We implicitly assume every such function can terminate—again, a safe approximation.

Next, we create an automaton that models the control and communication behavior for each function. Figure 4 shows automata for the three tasks (main, T_1, and T_2) of Figure 3. For each task, we build a deterministic finite state automaton whose edges represent choices, typically to communicate. The states are labeled by program counter values and the transitions by channel names. Each

automaton has a unique final state, which we draw as a double box. There is a transition from every terminating state to this final state labeled with a dummy channel that indicates such a transition. An automaton has only one final state but can have multiple terminating states. In the T_1 of Figure 3, 1 is the terminating state, 3 is the final state, and they are connected by τ_1, which is like a classical ε transition. An ε transition would make the automaton nondeterministic. Therefore we create this dummy channel τ_1, that is unique to T_1, and therefore allow T_1 to freely move from state 1 to state 3 without having to synchronize with any other another task.

The main function has a dummy π_{m1} transition from its start to the entry of state 2 ($T_1 \| T_2$), which represents the *par* statement in *main*. In general, we create a dummy channel for every *par* in the program.

Figure 5(a) shows the product of T_1 and T_2—an automaton that represents the combined behavior of T_1 and T_2. We constructed Figure 5(a) as follows. We start with state (program counter) values $(1,1)$. At this point, T_1 can communicate on a and move to state 2. Therefore we have an arc from $(1,1)$ to $(2,1)$ labeled by a. Similarly, T_2 can communicate on c and move to its state 2. From state $(1,1)$ it is not possible to communicate on b because only T_1 is ready to communicate, not T_2 (T_2 is also connected to b). Also at state $(1,1)$, T_1 can terminate by taking the transition τ_1 and moving to $(3,1)$.

From state $(3,1)$, T_2 can transition first to state $(3,2)$ by communicating on channel c and then to state $(3,3)$ by communicating on b; these transitions do not change the state of T_1 because it has already terminated.

From $(2,1)$, T_2 can communicate on c and change the state to $(2,2)$. Similarly from $(1,2)$, T_1 can communicate on a and move to $(2,2)$. In state $(1,2)$ it is also possible to communicate on b, since both tasks are ready. Therefore, we have an arc b from $(1,2)$ to $(2,3)$. Since T_1 may also choose to terminate in state $(1,2)$, there is an arc from $(1,2)$ to $(3,2)$ on τ_1. Other states follow similar rules.

To determine which channels may share buffers, we consider all states that have two or more outgoing edges. For example, in Figure 5(a), state $(1,1)$ has two outgoing transitions on a and c. Either of them can fire. In other words, this is a case where the program may choose to communicate on either a or c, meaning the contents of both of these buffers are needed at this point. Thus we conclude buffers for a and c may not share memory. We prove this formally later in the paper.

From Figure 3, it is evident that a and b can never occur together because T_1 sequentializes them. However, since state $(1,2)$ has outgoing transitions on a and b, our algorithm concludes that a and b can occur together. However, they actually can not. We draw this conclusion because our algorithm does not differentiate between scheduling choices and control flow choices (i.e., due to conditionals such as *if* and *while*). By doing this we are only adding extra behavior to the system and disregarding pairs of channels whose buffers actually could be shared. This is not a big disadvantage because our analysis remains safe. For this example our algorithm only allows b and c to share buffers.

Figure 5(b) is obtained by inlining the automaton for $T_1 \cdot T_2$—Figure 5(a)—within M. This represents the entire program in Figure 3. Since the *par* call is blocking, inlining $T_1 \cdot T_2$ within M is safe. We replaced state 2 of Figure 4(a) with Figure 5(a) to obtain Figure 5(b). The conclusions are the same as that of Figure 5(a)—only b and c can share buffers.

4. Merging Tasks

In this section, we use notation from automata theory to formalize the merging of two tasks. We show our algorithm does not generate any false negatives and is therefore safe.

DEFINITION 1. *A* deterministic finite automaton T *is a 5-tuple* $(Q, \Sigma, \delta, q, f)$ *where* Q *is the set of states,* Σ *is the set of channels,* $q \in Q_1$ *is the initial state,* $f \in Q$ *is the final state, and* $\delta \subseteq Q \times \Sigma \to Q$ *is the partial transition function.*

DEFINITION 2. *If* T_1 *and* T_2 *are automata, then the* composed automaton $T_1 \cdot T_2 = (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta_{12}, \langle q_1, q_2 \rangle, \langle f_1, f_2 \rangle)$*, where, for* $\langle p_1, p_2 \rangle \in Q_1 \times Q_2$ *and* $a \in \Sigma_1 \cup \Sigma_2$*,*

$$\delta_{12}(\langle p_1, p_2 \rangle, a) = \begin{cases} \langle \delta_1(p_1,a), \delta_2(p_2,a) \rangle & \text{if } a \in \Sigma_1 \text{ and } a \in \Sigma_2; \\ \langle \delta_1(p_1,a), p_2 \rangle & \text{if } a \in \Sigma_1 \text{ and } (a \notin \Sigma_2 \text{ or } p_2 = f_2); \\ \langle p_1, \delta_2(p_2,a) \rangle & \text{if } a \in \Sigma_2 \text{ and } (a \notin \Sigma_1 \text{ or } p_1 = f_1); \\ \text{undefined} & \text{otherwise;} \end{cases}$$

is the transition rule for composition.

In general, if T_1 has m states and T_2 has n, then the product $T_1 \cdot T_2$ can have at most mn states. The states are labeled by a tuple composed of the program counter values of the individual tasks. Each state can have at most k outgoing edges, where k is the total number of channels. Consequently, the total number of edges in the graph can at most be mnk (k accounts for the extra τ and π channels—one extra channel per task and one per *par*).

Below, we demonstrate that the order in which automata are composed does not matter, although the state labels will be different. First, we define exactly what we mean for two automata to be equivalent.

DEFINITION 3. *Two automata* $T_1 = (Q_1, \Sigma_1, \delta_1, q_1, f_1)$ *and* $T_2 = (Q_2, \Sigma_2, \delta_2, q_2, f_2)$ *are* equivalent *(written* $T_1 \equiv T_2$*) if and only if* $\Sigma_1 = \Sigma_2$ *and there exists a bijective function* $b : Q_1 \to Q_2$ *such that* $q_2 = b(q_1)$*,* $f_2 = b(f_1)$*, and for every* $p \in Q_1$ *and* $a \in \Sigma_1$*, either both* $\delta_1(p,a)$ *and* $\delta_2(b(p),a)$ *are defined and* $\delta_2(b(p),a) = b(\delta_1(p,a))$ *or both are undefined.*

LEMMA 1. *Composition is commutative:* $T_1 \cdot T_2 \equiv T_2 \cdot T_1$.

PROOF By definition, $T_1 \cdot T_2 = (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta_{12}, \langle q_1, q_2 \rangle, \langle f_1, f_2 \rangle)$ and $T_2 \cdot T_1 = (Q_2 \times Q_1, \Sigma_2 \cup \Sigma_1, \delta_{21}, \langle q_2, q_1 \rangle, \langle f_2, f_1 \rangle)$. We claim $b(\langle p_1, p_2 \rangle) = \langle p_2, p_1 \rangle$ is a suitable bijective function.

First, note that $\Sigma_1 \cup \Sigma_2 = \Sigma_2 \cup \Sigma_1$, $\langle q_2, q_1 \rangle = b(\langle q_1, q_2 \rangle)$, and $\langle f_2, f_1 \rangle = b(\langle f_1, f_2 \rangle)$.

Next,

$$\begin{aligned} &\delta_{21}(b(\langle p_1, p_2 \rangle), a) \\ &= \delta_{21}(\langle p_2, p_1 \rangle, a) \\ &= \begin{cases} \langle \delta_2(p_2,a), \delta_1(p_1,a) \rangle & \text{if } a \in \Sigma_2 \text{ and } a \in \Sigma_1; \\ \langle \delta_2(p_2,a), p_1 \rangle & \text{if } a \in \Sigma_2 \text{ and } (a \notin \Sigma_1 \text{ or } p_1 = f_1); \\ \langle p_2, \delta_1(p_1,a) \rangle & \text{if } a \in \Sigma_1 \text{ and } (a \notin \Sigma_2 \text{ or } p_2 = f_2); \\ \text{undefined} & \text{otherwise;} \end{cases} \\ &= b \left(\begin{cases} \langle \delta_1(p_1,a), \delta_2(p_2,a) \rangle & \text{if } a \in \Sigma_1 \text{ and } a \in \Sigma_2; \\ \langle p_1, \delta_2(p_2,a) \rangle & \text{if } a \in \Sigma_2 \text{ and } (a \notin \Sigma_1 \text{ or } p_1 = f_1); \\ \langle \delta_1(p_1,a), p_2 \rangle & \text{if } a \in \Sigma_1 \text{ and } (a \notin \Sigma_2 \text{ or } p_2 = f_2); \\ \text{undefined} & \text{otherwise;} \end{cases} \right) \\ &= b\big(\delta_{12}(\langle p_1, p_2 \rangle, a)\big) \end{aligned}$$

Thus, $T_1 \cdot T_2 \equiv T_2 \cdot T_1$. □

LEMMA 2. *Composition is associative:* $(T_1 \cdot T_2) \cdot T_3 \equiv T_1 \cdot (T_2 \cdot T_3)$.

PROOF By definition, $(T_1 \cdot T_2) \cdot T_3 = ((Q_1 \times Q_2) \times Q_3, (\Sigma_1 \cup \Sigma_2) \cup \Sigma_3, \delta_{(12)3}, \langle\langle q_1, q_2\rangle, q_3\rangle, \langle\langle f_1, f_2,\rangle, f_3\rangle)$ and $T_1 \cdot (T_2 \cdot T_3) = (Q_1 \times (Q_2 \times Q_3), \Sigma_1 \cup (\Sigma_2 \cup \Sigma_3), \delta_{1(23)}, \langle q_1, \langle q_2, q_3\rangle\rangle, \langle f_1, \langle f_2, f_3\rangle\rangle)$. We claim $b(\langle\langle p_1, p_2\rangle, p_3\rangle) = \langle p_1, \langle p_2, p_3\rangle\rangle$ is a suitable bijective function.

First, note that $(\Sigma_1 \cup \Sigma_2) \cup \Sigma_3 = \Sigma_1 \cup (\Sigma_2 \cup \Sigma_3)$, $\langle q_1, \langle q_2, q_3\rangle\rangle = b(\langle\langle q_1, q_2\rangle, q_3\rangle)$, and $\langle f_1, \langle f_2, f_3\rangle\rangle = b(\langle\langle f_1, f_2\rangle, f_3\rangle)$.

Next,

$$
\begin{aligned}
&\delta_{1(23)}(b(\langle\langle p_1, p_2\rangle, p_3\rangle), a) \\
&\quad = \delta_{1(23)}(\langle p_1, \langle p_2, p_3\rangle\rangle, a) \\
&\quad = \begin{cases}
\langle \delta_1(p_1,a), \langle \delta_2(p_2,a), \delta_3(p_3,a)\rangle\rangle & \text{if } a \in \Sigma_1 \text{ and } a \in \Sigma_2 \text{ and } a \in \Sigma_3; \\
\langle \delta_1(p_1,a), \langle \delta_2(p_2,a), p_3\rangle\rangle & \text{if } a \in \Sigma_1 \text{ and } a \in \Sigma_2 \text{ and } (a \notin \Sigma_3 \text{ or } p_3 = f_3); \\
\langle \delta_1(p_1,a), \langle p_2, \delta_3(p_3,a)\rangle\rangle & \text{if } a \in \Sigma_1 \text{ and } a \in \Sigma_3 \text{ and } (a \notin \Sigma_2 \text{ or } p_2 = f_2); \\
\langle \delta_1(p_1,a), \langle p_2, p_3\rangle\rangle & \text{if } a \in \Sigma_1 \text{ and } (a \notin \Sigma_2 \text{ or } p_2 = f_2) \text{ and } (a \notin \Sigma_3 \text{ or } p_3 = f_3); \\
\langle p_1, \langle \delta_2(p_2,a), \delta_3(p_3,a)\rangle\rangle & \text{if } a \in \Sigma_2 \text{ and } a \in \Sigma_3 \text{ and } (a \notin \Sigma_1 \text{ or } p_1 = f_1); \\
\langle p_1, \langle \delta_2(p_2,a), p_3\rangle\rangle & \text{if } a \in \Sigma_2 \text{ and } (a \notin \Sigma_1 \text{ or } p_1 = f_1) \text{ and } (a \notin \Sigma_3 \text{ or } p_3 = f_3); \\
\langle p_1, \langle p_2, \delta_3(p_3,a)\rangle\rangle & \text{if } a \in \Sigma_3 \text{ and } (a \notin \Sigma_1 \text{ or } p_1 = f_1) \text{ and } (a \notin \Sigma_2 \text{ or } p_2 = f_2); \\
\text{undefined} & \text{otherwise;}
\end{cases} \\
&\quad = b\left(\begin{cases}
\langle\langle \delta_1(p_1,a), \delta_2(p_2,a)\rangle, \delta_3(p_3,a)\rangle & \text{if } a \in \Sigma_1 \text{ and } a \in \Sigma_2 \text{ and } a \in \Sigma_3; \\
\langle\langle \delta_1(p_1,a), \delta_2(p_2,a)\rangle, p_3\rangle & \text{if } a \in \Sigma_1 \text{ and } a \in \Sigma_2 \text{ and } (a \notin \Sigma_3 \text{ or } p_3 = f_3); \\
\langle\langle \delta_1(p_1,a), p_2\rangle, \delta_3(p_3,a)\rangle & \text{if } a \in \Sigma_1 \text{ and } a \in \Sigma_3 \text{ and } (a \notin \Sigma_2 \text{ or } p_2 = f_2); \\
\langle\langle \delta_1(p_1,a), p_2\rangle, p_3\rangle & \text{if } a \in \Sigma_1 \text{ and } (a \notin \Sigma_2 \text{ or } p_2 = f_2) \text{ and } (a \notin \Sigma_3 \text{ or } p_3 = f_3); \\
\langle\langle p_1, \delta_2(p_2,a)\rangle, \delta_3(p_3,a)\rangle & \text{if } a \in \Sigma_2 \text{ and } a \in \Sigma_3 \text{ and } (a \notin \Sigma_1 \text{ or } p_1 = f_1); \\
\langle\langle p_1, \delta_2(p_2,a)\rangle, p_3\rangle & \text{if } a \in \Sigma_2 \text{ and } (a \notin \Sigma_1 \text{ or } p_1 = f_1) \text{ and } (a \notin \Sigma_3 \text{ or } p_3 = f_3); \\
\langle\langle p_1, p_2\rangle, \delta_3(p_3,a)\rangle & \text{if } a \in \Sigma_3 \text{ and } (a \notin \Sigma_1 \text{ or } p_1 = f_1) \text{ and } (a \notin \Sigma_2 \text{ or } p_2 = f_2); \\
\text{undefined} & \text{otherwise;}
\end{cases}\right) \\
&\quad = b\big(\delta_{(12)3}(\langle\langle p_1, p_2\rangle, p_3\rangle, a)\big)
\end{aligned}
$$

Thus, $(T_1 \cdot T_2) \cdot T_3) \equiv T_1 \cdot (T_2 \cdot T_3)$. □

LEMMA 3. $T_1 \cdot T_2 \cdot T_3 \cdots T_n \equiv (((T_1 \cdot T_2) \cdot T_3) \cdots) \cdot T_n$

PROOF Since the composition is commutative and associative, we can build the entire system incrementally by composing two tasks at a time. □

LEMMA 4. *The outgoing transitions from a given state represent every possible rendezvous that can occur at that particular state.*

PROOF According to the definition of δ, we add an outgoing edge to a state for every rendezvous that can happen immediately after that state.

Multiple outgoing arcs from a state may represent choices due to control statements (such as *if* or *while*). $\delta(p_1, a) = q_2$ and $\delta(p_1, b) = q_2$, then we have two outgoing choices due to control flow.

On the other hand, a scheduling choice may occur when composing two tasks. A scheduling choice occurs when the ordering between two rendezvous is unknown. This happens when two different pairs of tasks can rendezvous on two different channels at the same time.

Suppose $a \in \Sigma_1$ and $a \notin \Sigma_2$ and $\delta_1(p_1, a) = q_1$, and if $b \in \Sigma_2$ and $b \notin \Sigma_1$ and $\delta_2(p_2, b) = q_2$, then $\delta_{12}(\langle p_1, p_2\rangle, a) = \langle q_1, p_2\rangle$ and $\delta_{12}(\langle p_1, p_2\rangle, b) = \langle p_1, q_2\rangle$. Thus, for every possible scheduling choice, we have an outgoing edge from the given state.

The absence of any choice due to control or scheduling will leave it with either one or zero outgoing arcs. Consequently, the outgoing transitions from a given state represent all possible rendezvous that can occur at that particular state. They represent both control flow and scheduling choices. □

A scheduling choice imposes no ordering among rendezvous, therefore allowing the possibility of the rendezvous to happen at the same time.

THEOREM 1. *Two channels a and b can share buffers if, $\forall p$, at most one of $\delta(p,a)$ and $\delta(p,b)$ is defined, but not both.*

PROOF Suppose a and b can rendezvous at the same time and if p_1 represents the state of the program counter just before the rendezvous, then by Lemma 4 we have two outgoing arcs from p_1: $\delta(p_1, a) = q_1$ and $\delta(p_1, b) = q_2$

Consequently, for $\exists p$ both $\delta(p,a)$ and $\delta(p,b)$ exists. Conversely, if $\forall p$ at most one of $\delta(p,a)$ and $\delta(p,b)$ exists, then we can safely say that a and b can share buffers. □

Our algorithm does not differentiate between control flow choices (e.g., due to *if* or *while*) and scheduling choices (due to partial ordering of rendezvous). Both kinds of choices produce states having multiple outgoing arcs. We conclude that arcs going out from the same state cannot share buffers. The multiplicity can be contributed only by control choices leading to false positives, but our system is safe; whenever we are unsure, we do not allow sharing.

5. Tackling State Space Explosion

If two tasks communicate infrequently, there is a possibility that the number of states in the product machine will grow too large to deal with. We address this by introducing a threshold, which limits the stack depth our recursive product machine composition function may use, and corresponds to the longest simple path in the product machine. If we reach the threshold, we stop and treat separately the two tasks being composed.

This heuristic, which we chose because our implementation was running out of stack space on certain complex examples, has the advantage of applying exactly when we are unlikely to find opportunities to share buffer memory. Tightly coupled tasks tend to have small state spaces—these are exactly those that allow buffer memory to be shared. Loosely coupled tasks by definition run nearly independently and thus the communication patterns of most pairs of channels are uncontrolled, eliminating the chance to share buffers between them.

Algorithm 1 shows the composition algorithm. It recursively composes two states p_1 and p_2. The *depth* variable is initialized

to 0 and incremented whenever successor states are composed. Whenever *depth* exceeds the threshold, we declare failure.

Algorithm 1 compose(p_1, p_2, Σ_1, Σ_2, *depth*, *threshold*)

```
1: if depth ≤ threshold then
2:   for all a ∈ Σ1 ∪ Σ2 do
3:     ⟨q1,q2⟩ = δ(⟨p1,p2⟩,a)
4:     if ⟨q1,q2⟩ ∉ hash then
5:       Add ⟨q1,q2⟩ to hash
6:       compose(q1, q2, Σ1, Σ2, depth+1, threshold)
7:     end if
8:   end for
9: else
10:   print "Threshold exceeded"
11: end if
```

We draw conclusions about local channels (whose scope has been completely explored) and we remain silent about the others. We make safe conclusions even when other channels have not been completely explored.

THEOREM 2. *If our algorithm concludes that two channels a and b can share buffers after abstracting away channel c, then a and b can still share buffers in the presence of c.*

PROOF If a and b can share buffers, then there is a sequential ordering between them. By SHIM semantics (Edwards and Tardieu 2006b), introduction of a new channel can create ordering between two channels that are not ordered, but can never disrupt an existing sequential ordering. Therefore, if our algorithm concludes that two buffers can share channels, the introduction of a new channel does not affect the conclusion. □

We conclude that two channels can share buffers only if two conditions hold: the two channels have been explored completely and every state has at most one of the two channels in its outgoing edge set.

We take a bottom-up approach while merging groups of tasks. Tasks in a (preprocessed) SHIM program have a tree structure. We merge the leaf tasks of this tree before merging their parents. We stop merging when all tasks have exceeded the threshold, or if the complete program has been merged. This approach works nicely because it allows us to stop whenever we run out of time or space without ruining safety.

6. Buffer Allocation

Our static analysis algorithm produces a set S that contains pairs of channels that can share buffers. Let S' be the complement of this set. We represent it as a graph: channels represent vertexes and for every pair $\langle c_i, c_j \rangle \in S'$, we draw an edge between c_i and c_j. Two adjacent vertexes cannot share buffers. Every node has a weight, which corresponds to the size of the channel.

Minimizing buffer memory consumption, therefore, reduces to the weighted vertex covering problem (Malaguti et al. 2008; Malaguti 2008). It is defined as follows: A graph G is colored with p colors such that no two adjacent vertexes are of the same color. We denote the maximum weight of a vertex colored with color i as $\max(i)$, and we need to find a coloring such that $\sum_{i=1}^{p} \max(i)$ is minimum. The problem is NP-hard.

We use a greedy first-fit algorithm to get an approximate solution. Let G be a list of groups. Initially G is empty. We order the channels in non-increasing order of the buffer sizes, then add the channels one by one to the first non-conflicting group in G. If there is no such group, we create a new group in G and add the channel to this newly created group. A group is defined to be non-conflicting if the channel to be added can share its buffer with every channel already in the group. Channels in the same group can share buffers. This algorithm runs in polynomial time but does not guarantee an optimal solution.

7. Experimental Results

We implemented our algorithm and ran it on various SHIM programs. Table 1 lists the results on running the experiments on a 3 GHz Pentium 4 Linux machine with 1 GB RAM. For each example, the columns list the number of lines of code in the program, the total number of channels it uses, the number of tasks that take part in communication (i.e., excluding any functions that perform no communication), the number of bytes of buffer memory saved by applying our algorithm, what percentage this is of overall buffer memory, the time taken for analysis (including compilation, abstraction, verification, and grouping buffers), and the number of states our algorithm explored. For these experiments, we set the threshold to 8000.

Source-Sink is a simple example of a FIFO with two processes: one that passes data and the other that prints the results through an output channel. Pipeline is a modification of source-sink that uses two buffer processes in between the input and output process.

Bitonic Sort uses multiple tasks for that compare and shuffle pairs of data values. They interact through thirteen channels.

The Prime Number Sieve example has bounded recursion and uses the technique of Edwards and Zeng (2008) to remove it.

The Berkeley example has communication patterns that are data dependent. We abstract away the data, making it simpler to analyze.

Framebuffer contains a line drawing task that drives a 640×480 video framebuffer. The communication pattern is complicated.

FFT takes an audio file as input, divides it into 1024-sample blocks performs fixed-point FFT on each block, then does an inverse FFT. It uses the largest buffers of all the example programs.

The JPEG decoder is one of the largest applications currently written in SHIM. It has multiple IDCT processors that run concurrently on groups of macroblocks passed around through buffers.

The FIR filter is a parallel filter with twenty-eight channels. It takes about thirteen seconds to analyze this program and the number of states explored is about eighty thousand. Since this was one of the more challenging examples for our algorithm, we tried varying the threshold. Table 2 summarizes our results. As expected, the number of visited states increases as we increase the threshold. With a threshold of 1000, we hardly explore the program, but higher thresholds let us explore more. When the threshold reaches 5000, we have explored enough of the system to begin to find opportunities for sharing buffer memory, even though we have not explored the system completely.

Experimentally, we find that the analysis takes less than a minute for modestly large programs and that we can reduce buffer space by 60% and therefore considerable amount of PPE's memory for examples like the bitonic sort and the prime number sieve.

8. Related Work

Many memory reduction techniques exist for embedded systems. de Greef et al. (1997) reduce array storage in a sequential program by reusing memory. Their approach has two phases: they internally reduce storage for each array, then globally try to share arrays. By contrast, our approach looks for sharing opportunities globally on communication buffers in a concurrent setting.

StreamIt (Thies et al. 2002) is a deterministic language like SHIM. Sermulins et al. (2005) present cache aware optimizations by exploiting communication pattern in StreamIt programs. Their aim is to improve instruction and data locality at the cost of data buffer size. We have the opposite goal of reducing buffer sizes.

Chrobak et al. (2001) schedule tasks in a multiprocessor environment to minimize maximum buffer size. Our algorithm does

Example	Lines	Channels	Tasks	Bytes Saved	Buffer Reduction	Runtime	States
Source-Sink	35	2	11	4	50 %	0.1 s	394
Pipeline	35	5	9	16388	25	0.1	68
Bitonic Sort	35	5	13	12	60	0.1	135
Prime Number Sieve	40	5	16	12	60	0.5	122
Berkeley	40	3	11	4	33.33	0.6	285
FIR Filter	110	28	28	52	46.43	13.8	74646
Framebuffer	185	11	16	28	0.002	1.3	15761
FFT	230	14	15	344068	50	0.6	3750
JPEG Decoder	1020	7	15	772	50.13	1.8	517

Table 1. Experimental results with the threshold set to 8000

Threshold	Bytes Saved	Buffer Reduction	Runtime	States
2000	0	0 %	0.6 s	10024
3000	0	0	1.5	23530
4000	0	0	3.4	51086
5000	52	46.43	12.4	70929
6000	52	46.43	12.8	72101
7000	52	46.43	13.5	73433
8000	52	46.43	13.8	74646

Table 2. Effect of threshold on the FIR filter example

not add scheduling constraints to the problem: it reduces the total buffer size with affecting the schedule, and thereby not affecting the overall speed.

The work of Murthy and Bhattacharyya (2000, 2001, 2004, 2006) and Teich et al. (1998) is closest to ours. They describe several algorithms for merging buffers in signal processing systems that use synchronous data flow models (Lee and Messerschmitt 1987). Govindarajan et al. (2002) minimize buffer space while executing at the optimal computation rate in dataflow networks. They cast this as a linear programming problem and solve it. Sofronis et al. (2006) propose an optimal buffer scheme with a synchronous task model as basis. These papers revolve around minimizing buffers in a synchronous setting; our work solves similar problems in an asynchronous setting. Our approach finds if there is an ordering between rendezvous of different channels based on the product machine. We believe that our algorithm works on a richer set of programs.

Lin (1998a,b) talks about an efficient compilation process of programs that have communication constructs similar to SHIM. He uses Petri nets to model the program and uses loop unrolling techniques. We did not attempt this approach because loop unrolling would cause the state space to explode even for small SHIM programs.

Static verification methods already exist for SHIM. For example, Vasudevan and Edwards (2008) build a synchronous system to find deadlocks in a SHIM program. They make use of the fact that for a particular input sequence, if a SHIM program deadlocks under one schedule it will deadlock under any other. By contrast, the property we check in this paper is not schedule-independent: two channels may rendezvous at the same time under one schedule but may not under another schedule. This makes our problem more challenging.

Edwards and Tardieu (2006a) describe a partial evaluation method that combines multiple concurrent processes to produce sequential code. Again, they make use of the scheduling independence property by expanding one task at a time until it terminates or blocks on a channel. On the other hand, we expand all possible communications from a given state and therefore forcing us to consider all tasks that can communicate from that state, rather than a single task.

9. Conclusions

We presented a static buffer memory minimization technique for the SHIM concurrent language. We obtain the partial order between communication events on channels by forming the product machine of potentially all tasks in a program.

We remove bounded recursion and expand each SHIM program into a tree of tasks and use sound abstractions to construct for each task an automaton that performs communication. Then we use the merging rules to combine tasks.

We abstract away data and computation from the program and only maintain parallel, communication and branch structures. We abstract away the data-dependent decisions formed by conditionals and loops. We do not differentiate between scheduling choices and conditional branches. This may lead to to false positives: our technique can discard pairs even though it can share buffers. However, our experimental results suggest this is not a big disadvantage and in any case our technique remains safe.

Our algorithm can be practically applied to the SHIM compiler that generates code for the Cell Broadband Engine. For instance, we can save 344KB of the PPE's memory for the FFT example.

We reduce memory without affecting the run-time schedule or performance. By sharing, two or more buffer pointers point to the same memory location and this can be done at compile-time during the code-generation phase.

To avoid state space explosion, we introduced a threshold for limiting the recursion depth our algorithm must handle. We plan to look into more modular techniques that allow a set of tasks to be analyzed independent of the remaining sets.

We currently ignore SHIM's exceptions (Tardieu and Edwards 2006b). Exceptions in SHIM provide a convenient way to terminate peer tasks and they are deterministic in behavior. We also plan to consider exceptions in the future.

References

Marek Chrobak, János Csirik, Csanád Imreh, John Noga, Jiri Sgall, and Gerhard J. Woeginger. The buffer minimization problem for multipro-

cessor scheduling with conflicts. In *ICALP '01: Proceedings of the 28th International Colloquium on Automata, Languages and Programming,*, pages 862–874, London, UK, 2001. Springer-Verlag.

Eddy de Greef, Francky Catthoor, and Hugo de Man. Array placement for storage size reduction in embedded multimedia systems. In *ASAP '97: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, page 66, Washington, DC, USA, 1997. IEEE Computer Society.

Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 37–44, Jersey City, New Jersey, September 2005.

Stephen A. Edwards and Olivier Tardieu. Efficient code generation from SHIM models. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 125–134, Ottawa, Canada, June 2006a.

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 2006b.

Stephen A. Edwards and Jia Zeng. Static elaboration of recursion for concurrent software. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation (PEPM)*, San Francisco, California, January 2008.

Stephen A. Edwards, Nalini Vasudevan, and Olivier Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 1498–1503, Munich, Germany, March 2008.

R. Govindarajan, Guang R. Gao, and Palash Desai Y. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *Journal of VLSI Signal Processing Systems*, 31(3):207–209, July 2002.

C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, New Jersey, 1985.

James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5): 589–604, July/September 2005.

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.

Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, May-June 2006.

Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

Bill Lin. Efficient compilation of process-based concurrent programs without run-time scheduling. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 211–217, Paris, France, February 1998a.

Bill Lin. Software synthesis of process-based concurrent programs. In *Proceedings of the 35th Design Automation Conference*, pages 502–505, San Francisco, California, June 1998b.

E. Malaguti. The vertex coloring problem and its generalizations. *4OR: A Quarterly Journal of Operations Research*, 2008.

E. Malaguti, M. Monaci, and P. Toth. Models and heuristic algorithms for a weighted vertex coloring problem. *Journal of Heuristics*, 2008.

Praveen K. Murthy and Shuvra S. Bhattacharyya. Systematic consolidation of input and output buffers in synchronous dataflow specifications. *IEEE Workshop on Signal Processing Systems (SiPS)*, pages 673–682, 2000.

Praveen K. Murthy and Shuvra S. Bhattacharyya. Shared buffer implementations of signal processing systems using lifetime analysis techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):177–198, February 2001.

Praveen K. Murthy and Shuvra S. Bhattacharyya. Buffer merging—a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Transactions on Design Automation of Electronic Systems*, 9(2):212–237, April 2004.

Praveen K. Murthy and Shuvra S. Bhattacharyya. *Memory Management for Synthesis of DSP Software*. CRC Press, Inc., Boca Raton, FL, USA, 2006.

Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache aware optimization of stream programs. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 115–126, New York, NY, USA, 2005. ACM.

Christos Sofronis, Stavros Tripakis, and Paul Caspi. A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 21–33, New York, NY, USA, 2006. ACM.

Olivier Tardieu and Stephen A. Edwards. R-SHIM: Deterministic concurrency with recursion and shared variables. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, page 202, Napa, California, July 2006a.

Olivier Tardieu and Stephen A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 142–151, Seoul, Korea, October 2006b.

Jürgen Teich, Eckart Zitzler, and Shuvra S. Bhattacharyya. Buffer memory optimization in dsp applications — an evolutionary approach. In *Proceedings of Parallel Problem Solving from Nature (PPSN)*, pages 885–896, London, UK, 1998. Springer-Verlag.

William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the International Conference on Compiler Construction (CC)*, volume 2304 of *Lecture Notes in Computer Science*, pages 179–196, Grenoble, France, April 2002.

Nalini Vasudevan and Stephen A. Edwards. Static deadlock detection for the SHIM concurrent language. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, Anaheim, California, June 2008.

Nalini Vasudevan and Stephen A. Edwards. Celling SHIM: Compiling deterministic concurrency to a heterogeneous multicore. In *Proceedings of the Symposium on Applied Computing (SAC)*, volume III, pages 1626–1631, Honolulu, Hawaii, March 2009.

Nalini Vasudevan, Satnam Singh, and Stephen A. Edwards. A deterministic multi-way rendezvous library for Haskell. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, Miami, Florida, April 2008.