# Buffer Sharing in Rendezvous Programs

Nalini Vasudevan, *Student Member, IEEE,* and Stephen A. Edwards, *Senior Member, IEEE*

***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. We present a memory reduction technique for CSP-style rendezvous communication, which we apply to our deterministic concurrent programming language SHIM. We focus on reducing memory consumption by sharing communication buffers among tasks. 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 are never occupied simultaneously. Experimentally, we find our technique runs quickly on modest-sized programs and can sometimes reduce memory requirements by half.**

***Index Terms*—Concurrency, SHIM, Static Analysis, Buffers, Optimization**

## I. 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 [1]. It is closely related to some of the techniques we used to statically detect deadlocks [2], but we address a different problem here.

SHIM is an asynchronous concurrent language that is scheduling-independent: its input/output behavior is not affected by any non-deterministic scheduling choices taken by its runtime environment due to processor speed, the operating system, scheduling policy, etc. A SHIM program consists of sequential tasks that synchronize when they want to communicate. The language is a subset of Kahn networks [3] (to ensure determinism) that employs the rendezvous of Hoare's CSP [4] for communication to keep its behavior tractable.

SHIM processes communicate through channels (Figure 1). Every task maintains its own local variables, and in most SHIM implementations, any communication involves copying to and reading from a shared memory location. The sequence of symbols transmitted over each channel is deterministic but the relative order of symbols on different channels is generally undefined. However, if we can determine that the relative order of symbols on a pair of channels is such that they never interfere, we can safely share the buffers for the channels. If we cannot find such an ordering, we conclude that the pair cannot share memory.

N. Vasudevan and S. A. Edwards are with the Department of Computer Science, Columbia University, New York, NY, 10027 USA e-mail: (naliniv@cs.columbia.edu, sedwards@cs.columbia.edu).

Manuscript received Dec 10, 2009; revised April 1, 2010.

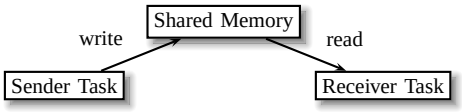


Fig. 1. The channel structure

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 [5]. The heterogeneous Cell processor [6] 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. An SPE can communicate with the PPE through mailboxes [7].

We wish to reduce memory used by the PPE by overlapping buffers of different channels. Our static analyzer does live range analysis on the communication channels and determines pairs of buffers that are never live at the same time. We demonstrate in Section VII 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 II), how we model its behavior to analyze buffer usage (Section III), how we compose models of SHIM tasks to build a product machine for the whole program (Section IV), how we avoid state explosion (Section V), and how we use these results to reduce buffer memory usage (Section VI). We present experimental results in Section VII and the application of our algorithm to Cell Programs in Section VIII. We discuss related work in Section X and conclude in Section XI.

## II. The SHIM Programming Language

SHIM [1] is a C-like concurrent programming language whose tasks communicate exclusively through multi-way rendezvous channels. To the usual collection of C-like expressions and statements it adds two constructs: *par* for specifying concurrency, and *send* and *recv* for communication. *p par q* runs statements *p* and *q* in parallel and finishes when both *p* and *q* terminate. *Send c* and *recv c* are communication statements that synchronize on channel *c*. SHIM has no global or shared variables.

```
void main()
{
 chan int a, b;
 {         // Task 1
   send a = 6; // Send 6 on a (synchronize w/ 2)
               // a = 6 here
   recv b;     // Receive b   (synchronize w/ 2)
               // b = 8 here
 } par { // Task 2
   recv a;     // Receive a   (synchronize w/ 1)
               // a = 6 here
   send b = 8; // Send 8 on b (synchronize w/ 1)
               // b = 8 here
  }
}
```

Fig. 2. A SHIM program where two tasks communicate through two channels

```
void main()
{
  chan int a, b, c;
  {         // Task 1
    send a = 6;      // Send a       (synchronize w/ 2)
  } par { // Task 2
    recv a;          // Receive a    (synchronize w/ 1)
    send b = a + 1; // Send 7 on b (synchronize w/ 3)
  } par { // Task 3
    recv b;          // Receive b    (synchronize w/ 2)
    send c = b + 1; // Send 8 on c (synchronize w/ 4)
  } par { // Task 4
    recv c;          // Receive c    (synchronize w/ 3)
                     // c = 8 here
  }
}
```

Fig. 3. A SHIM program that illustrates the need for buffer sharing

In Figure 2, two SHIM tasks run concurrently within *main* and communicate on channels *a* and *b*. The *send a* in task 1 assigns 6 to *a* and waits for task 2 to receive the value. The tasks therefore rendezvous then continue to their next statements. Next, the two tasks rendezvous at *b*. There, task 1 receives the value 8 from task 2.

Small changes to this program can produce different behavior. If both tasks (statically) attempted to send on a channel, the compiler would reject the program. If statements *recv a* and *send b = 8* were interchanged, the program would deadlock.

Back ends of our SHIM compiler can generate C code for a variety of environments: shared-memory multiprocessors using the pthreads library [8], the IBM Cell Broadband Engine [5], and single-threaded processors that do not require thread support [9]. The SHIM model has also been implemented as a library for Haskell [10] and even hardware translation has been proposed [11].

The goal of our work is buffer sharing, which we illustrate using the program in Figure 3. Here, 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*. Finally, task 4 receives 8 on channel *c*. Communication on *a* cannot occur simultaneously with that of *b* because task 2 forces them to occur sequentially. Similarly communications on *b* and *c* are forced to be sequential by task 3. Communications on *a* and *c* cannot occur together because they are forced to be sequential 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 for this program by 66%. Although this only represents the savings of a few words in this example, SHIM communication channels often pass large objects such as arrays, in which case a 66% reduction can be substantial. Our experimental results in Section VII demonstrate this.

## III. Abstracting SHIM Programs

Our technique abstracts a SHIM program down to its communication patterns to identify situations in which buffers can be active simultaneously and thus not shared. We describe this below.

First, we assume that a SHIM program has no recursion. While the SHIM language allows it, we can use the techniques of Edwards and Zeng [12] to remove bounded recursion, which makes the program finite and renders 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 full state space of even a small program is usually too large; 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 our analysis will conclude two channels can 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. This choice proved to be wise. For the programs discussed in Section VII, we never lost any opportunity for sharing by assuming both sides of a branch are followed that an exact analysis would have enabled. Besides, it is impossible to predict at compile-time the exact behavior of branches that depend on program input.

### A. An Example

In Figure 4, the *main* function consists of two tasks that communicate through channels *a*, *b*, and *c*.

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 is 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, i.e., it is the only process that participates in a rendezvous on its channels. Thus, terminated processes do not cause other processes to deadlock.

```
void main() {
  chan int a, b, c;
  {          // Task 1
    for (int i = 0; i < 15; i++) {  //  state 1
      if (i % 2 == 0)
        send a = 5;
      else
        send b = 7;
                                    //  state 2
      send b = 10;
    }
                                    //  state 3
  } par {  // Task 2
                                    //  state 1
    send c = 13;
                                    //  state 2
    recv b;
                                    //  states 3 & 4
  }
}
```

Fig. 4. A (contrived) SHIM program with a loop, conditionals, and a task that terminates

At compilation time, the compiler dismantles the main function of Figure 4 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 4, communication on $b$ cannot occur simultaneously with that on c because $T_2$ forces the two communications to be sequential 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 our static deadlock detector [2]. It begins by removing bounded recursion and duplicating functions to force every call site to be unique. This has the potential of producing an exponential blow-up, but we have not observed this in practice because bounded recursion in SHIM programs usually generates structure rather that being algorithmic.

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

Next we disregard all functions that do not affect the communication 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 5 shows automata for the three tasks (main, $T_1$, and $T_2$) of Figure 4. For each task, we build a deterministic finite state automaton

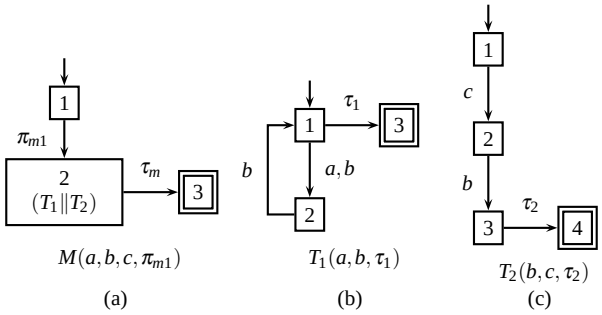


Fig. 5. Automata for (a) the main task and (b), (c) its subtasks

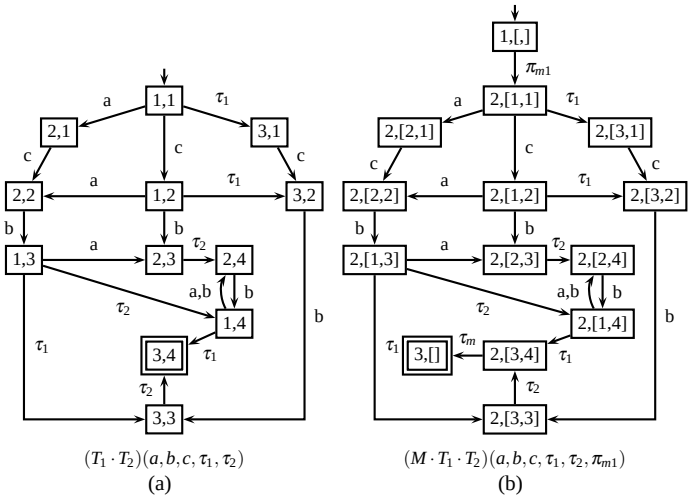


Fig. 6. Composing tasks in Figure 5: (a) Merging $T_1$ and $T_2$. (b) Inlining $T_1 \cdot T_2$ in $M$.

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 Figure 5(b), $T_1$'s state 1 is the terminating state, state 3 is the final state, and they are connected by $\tau_1$, which is like a classical $\varepsilon$ transition. However, $\varepsilon$ edges would make the automaton nondeterministic, so we instead create a dummy channel $\tau_1$ that is unique to $T_1$ and allow $T_1$ to move from state 1 to state 3 without having to synchronize.

The main function has a dummy $\pi_{m1}$ 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 6(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 6(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 $a$. 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 $\tau_1$ 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 $\tau_1$. 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 6(a), state $(1,1)$ has outgoing transitions on $a$ and $c$. Either of them can fire, so this is a case where the program may choose to communicate on either $a$ or $c$. This means the contents of both of these buffers are needed at this point, so we conclude buffers for $a$ and $c$ may not share memory. We prove this formally below.

From Figure 4, it is evident that $a$ and $b$ can never occur together because $T_1$ forces them to be sequential. 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 erroneous 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 6(b) is obtained by inlining the automaton for $T_1 \cdot T_2$—Figure 6(a)—within $M$. This represents the entire program in Figure 4. Since the *par* call is blocking, inlining $T_1 \cdot T_2$ within $M$ is safe. We replaced state 2 of Figure 5(a) with Figure 6(a) to obtain Figure 6(b). The conclusions are the same as that of Figure 6(a)—only $b$ and $c$ can share buffers.

## IV. 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, $\Sigma$ 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 $\tau$ and $\pi$ 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, the states are isomorphic. 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,

$$\begin{aligned} T_1 \cdot T_2 &= (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta_{12}, \langle q_1,q_2\rangle, \langle f_1,f_2\rangle) \text{ and} \\ T_2 \cdot T_1 &= (Q_2 \times Q_1, \Sigma_2 \cup \Sigma_1, \delta_{21}, \langle q_2,q_1\rangle, \langle f_2,f_1\rangle). \end{aligned}$$

We claim $b(\langle p_1,p_2\rangle) = \langle p_2,p_1\rangle$ is a suitable bijective function. First, note $\Sigma_1 \cup \Sigma_2 = \Sigma_2 \cup \Sigma_1$, $\langle q_2,q_1\rangle = b(\langle q_1,q_2\rangle)$, and $\langle f_2,f_1\rangle = b(\langle f_1,f_2\rangle)$.

Next,

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

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,

$$\begin{aligned} (T_1 \cdot T_2) \cdot T_3 &= ((Q_1 \times Q_2) \times Q_3, (\Sigma_1 \cup \Sigma_2) \cup \Sigma_3, \delta_{(12)3}, \\ &\qquad \langle\langle q_1,q_2\rangle,q_3\rangle, \langle\langle f_1,f_2,\rangle,f_3\rangle) \\ T_1 \cdot (T_2 \cdot T_3) &= (Q_1 \times (Q_2 \times Q_3), \Sigma_1 \cup (\Sigma_2 \cup \Sigma_3), \delta_{1(23)}, \\ &\qquad \langle q_1,\langle q_2,q_3\rangle\rangle, \langle f_1,\langle f_2,f_3\rangle\rangle). \end{aligned}$$

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), & \text{if } a\in\Sigma_1 \text{ and } a\in\Sigma_2 \text{ and}\\
\qquad \delta_3(p_3,a)\rangle\rangle & 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, & \text{if } a\in\Sigma_1 \text{ and } a\in\Sigma_2 \text{ and}\\
\qquad \delta_3(p_3,a)\rangle & 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 $\delta$, 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, thus allowing the possibility of two or more 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 some $p$, both $\delta(p,a)$ and $\delta(p,b)$ exist. Conversely, if for all $p$ at most one of $\delta(p,a)$ and $\delta(p,b)$ exists, we can safely say $a$ 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.

## V. 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 compute. We address this by introducing a threshold, which limits the stack depth of our recursive product machine composition procedure and corresponds to the longest simple path in the product machine. If we reach the threshold, we stop and treat the two tasks being composed as being separate (i.e., unable to share buffers with each other).

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 pattern of most pairs of channels are uncontrolled, eliminating the chance to share buffers between them.

**Algorithm 1** compose($p_1, p_2, \Sigma_1, \Sigma_2, depth, threshold$)

```
if depth > threshold then
    return "Threshold exceeded"
else
    for all a ∈ Σ1 ∪ Σ2 do
        ⟨q1, q2⟩ = δ(⟨p1, p2⟩, a)
        if ⟨q1, q2⟩ ∉ hash then
            Add ⟨q1, q2⟩ to hash
            compose(q1, q2, Σ1, Σ2, depth + 1, threshold)
```

| Example | Lines | Channels | Tasks | Bytes Saved | Buffer Reduction | Run Time | 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 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 | 12 | 10 | 286720 | 41.6 | 0.8 | 2192 |
| JPEG Dec. | 990 | 12 | 9 | 983040 | 55.55 | 1.5 | 2192 |

TABLE I
EXPERIMENTAL RESULTS WITH THE THRESHOLD SET TO 8000

| IDCT Tasks | Lines | Channels | Total Tasks | Bytes Saved | Buffer Reduction | Run time | States |
|---|---|---|---|---|---|---|---|
| 1 | 940 | 2 | 4 | 98304 | 33.33 % | 0.5 s | 26 |
| 2 | 950 | 4 | 5 | 196608 | 33.33 | 0.6 | 64 |
| 3 | 960 | 6 | 6 | 393216 | 44.44 | 0.7 | 158 |
| 4 | 970 | 8 | 7 | 589824 | 50 | 0.9 | 386 |
| 5 | 980 | 10 | 8 | 786432 | 53.33 | 1.2 | 928 |
| 6 | 990 | 12 | 9 | 983040 | 55.55 | 1.5 | 2192 |

TABLE II
BEHAVIOR OF THE JPEG DECODER WITH VARYING NUMBER OF IDCT TASKS (THRESHOLD SET TO 8000)

| FFT Tasks | Lines | Channels | Total Tasks | Bytes Saved | Buffer Reduction | Run time | States |
|---|---|---|---|---|---|---|---|
| 1 | 150 | 2 | 4 | 57344 | 50 % | 0.2 s | 25 |
| 2 | 160 | 4 | 5 | 114688 | 50 | 0.3 | 63 |
| 3 | 180 | 6 | 6 | 172032 | 50 | 0.4 | 158 |
| 4 | 200 | 8 | 7 | 172032 | 37.5 | 0.6 | 385 |
| 5 | 210 | 10 | 8 | 229376 | 40 | 0.7 | 927 |
| 6 | 230 | 12 | 9 | 286720 | 41.6 | 0.8 | 2192 |

TABLE III
BEHAVIOR OF FFT WITH VARYING NUMBER OF PROCESSING TASKS (THRESHOLD SET TO 8000)

Algorithm 1 is 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.

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 [11], introduction of a new channel can create ordering between two channels that are not ordered, but can never disrupt an existing sequential ordering unless it introduces a deadlock. Therefore, if our algorithm concludes that two buffers can share channels, introducing a new channel does not affect the conclusion since we assume deadlock-free programs. ■

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 that arises from nesting of *par* constructs. 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 allows us to stop whenever we run out of time or space without violating safety.

## VI. 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 vertices and for every pair $\langle c_i, c_j \rangle \in S'$, we draw an edge between $c_i$ and $c_j$. Two adjacent vertices cannot share buffers. Every node has a weight, which corresponds to the size of the channel.

Minimizing buffer memory consumption, therefore, reduces to the NP-hard weighted vertex coloring problem [13], [14]: a graph $G$ is colored with $p$ colors such that no two adjacent vertices 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.

We use a greedy first-fit algorithm as a heuristic. Let $G$ be a list of groups, initially empty. We order the channels in non-increasing order of 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.

## VII. EXPERIMENTAL RESULTS

We implemented our algorithm and ran it on various SHIM programs. Table I lists the results of 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

| 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 IV
EFFECT OF THRESHOLD ON THE FIR FILTER EXAMPLE

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, along with a number of intermediate stages. Pipeline is a modification of source-sink that uses two buffer processes in between the input and output process. The Pipeline example has larger buffers because it passes large amounts of data between stages.

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

The Prime Sieve example has bounded recursion that is removed as part of the compilation process [12].

The Berkeley example has data-dependent communication patterns. We abstract away the data, simplifying the analysis.

Framebuffer contains a line drawing task that drives a $640 \times 480$ video framebuffer. The framebuffer hardly gets any savings because no concrete data is passed among tasks. The tasks communicate with each other just through synchronization signals.

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.

For the JPEG and the FFT Examples, we created varying number of threads and measured the reduction in buffer memory; see Table II and Table III. For the JPEG decoder and FFT, we save upto 55% and 45% respectively of buffer memory.

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 IV 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 memory on the Cell processor for examples like the bitonic sort and the prime number sieve. We have also reported a subset of our findings in [15].

## VIII. APPLICATIONS

One concrete application of our technique is to reduce memory consumption in a distributed architecture like the Cell broadband engine [16], [6], [7]. Secondly, the output of our algorithm can be used as a strategy to distribute buffers.

### A. Optimizing Cell Programs

The Cell processor, one target of our SHIM compiler [5], uses a heterogeneous architecture consisting of a traditional 64-bit power processor element (PPE) with its own 32K L1 and 512K L2 caches coupled to eight synergistic processor elements (SPEs).

Each SPE is an 128-bit processor whose ALU can perform up to 16 byte operations in parallel. Each has 128 128-bit general-purpose (vector) registers, a 256K local store, but no cache. Each SPE provides high, predictable performance on vectors.

Cell programs use direct-memory access (DMA) operations to transfer data among the PPE and SPEs' memories. While addresses are global (i.e., addresses for the PPE's and each SPE's memories are distinct), this is not a shared memory model. Although direct SPE-to-SPE communication is possible, it is easier to implement a communication between an SPE and the PPE since handshaking is easy to implement using the Cell's hardware mailboxes.

Our Cell compiler [5] for SHIM uses multiple cores to provide task-level parallelism. Tasks communicate through dedicated buffers in the PPE. Since the communication buffers reside in the PPE, there is a possibility of sharing.

Figure 7 shows the block diagram of a JPEG decoding application with 4 IDCT processors and a possible mapping onto the Cell processor. The JPEG application is easily pipelined, at least partially: data is read (R) from the input stream, Huffman decoded (Huf), processed, an IDCT is applied, and finally written (W) to the output stream.

Both the read (R) and write (W) blocks must be executed sequentially. The I/O operations are performed by the PPE, therefore we dedicate PPE threads to read and write operations. The Huffman decoding blocks are executed in sequence but by an SPE. The IDCT blocks can run independently and therefore executed concurrently by different SPEs.

The tasks communicate with each other through channels. In Figure 7, *in*, *stripe1*, *stripe2*, *stripe3*, *stripe4*, *out1*, *out2*, *out3* and *out4* are communication channels. SPE 0 communicates with other SPEs through channels *stripe1*, *stripe2*, *stripe3* and *stripe4*. The communication buffers of these channels are located in the PPE. For example, SPE 1 writes to *stripe1* in PPE's memory and then SPE 2 reads the value from it. The

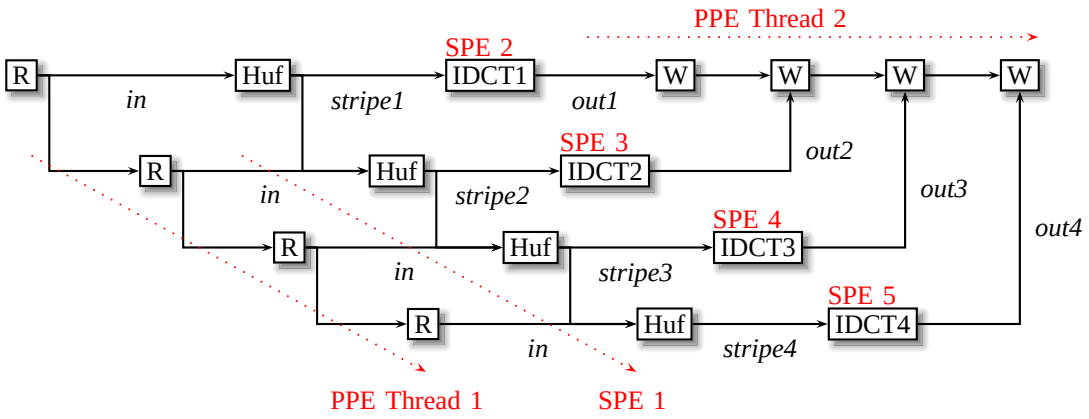


Fig. 7. JPEG block diagram with four IDCT processors

Huffman decoder first communicates *stripe1* to SPE 2, then *stripe2* to SPE 3 followed by *stripe3* to SPE 4 and finally *stripe4* to SPE 5. Since the communication is ordered, the four channels can use the same buffer space in the PPE. Each stripe occupies about 200KB of PPE memory, therefore our technique saves a total of about 600KB of space.

### B. Distributed Buffer Allocation

For many distributed applications, buffers do not reside in the same memory. This is advantageous in two ways. Firstly, a single node cannot capacitate all the buffers. Secondly, a distributed memory allocation is more fault-tolerant than a centralized memory allocation. Our memory reduction algorithm can be used to find ways of distributing buffers over different nodes. Secondly, we can specify ways of distributing tasks among nodes by determining what buffers they access.

Consider a program that has four channels *a*, *b*, *c* and *d* that require buffers of equal sizes. Suppose our algorithm finds that *a* and *b* can share channels, then we put *a* and *b* on the same node. We can also tie to this node, the tasks that access *a* and *b* frequently. This provides a strategy for distributed memory allocation that optimizes memory and locality.

## IX. Limitations and Applicability to Other Languages

Our technique makes some key assumptions about the structure and behavior of programs that limits its applicability. While the SHIM language has most of these limitations, our technique could be adapted to work in other settings.

One major assumption is that the call graph of any program is a tree, enabling us to statically determine all the tasks and the channels to which each is connected. Thus, our approach cannot be directly applied to a program that dynamically creates tasks or changes its connectivity. While plenty of programs do have such a dynamic nature, many are static in the sense we assume.

Secondly, SHIM does not support pointers, implying channels and tasks are determined at compile time. For languages that support references to channels and tasks, a good alias analysis step would have to be added to our technique to make it work.

We implemented the SHIM communication model as a library [10] in Haskell, and we believe such a library could be written for many concurrent languages. For a program that use our library, it would be necessary to first verify the program is using the API correctly before applying our technique, but this should be possible.

```
chan int x, a, b, c;
{
  x = 1;
  send x;
} par {
  recv x;
  if (x > 5) {
    recv a;
    recv c;
  } else {
    recv a;
    recv b;
    recv c;
  }
}
```

Fig. 8. A code fragment that is affected by branch predicate abstraction

For efficiency, our analysis completely ignores data and assumes that both paths of a branch can be taken, but this sometimes leads to over-conservative results. Consider Figure 8. Here, only the *else* branch is taken since the received *x* is 1 and therefore less than 5. This code sequentializes channels *a*, *b* and *c*. However, our analysis assumes that the *if* part of the branch can also be taken, which sequentializes only *a* and *c*. Therefore, our analysis erroneously concludes that *b* cannot share buffers with *a* and *c*. In practice, such patterns do not appear very often. In particular, ignoring data did not negatively influence the results for any of the examples in Section VII, but our technique could be improved by performing a global analysis to obtain relations among branch predicates.

## X. Related Work

Many memory reduction techniques exist for embedded systems. Greef et al. [17] 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 [18] is a deterministic language like SHIM. Sermulins et al. [19] present cache aware optimizations that exploit communication pattern in StreamIt programs. They aim to improve instruction and data locality at the cost of data buffer size. Instead, we try to reduce buffer sizes.

Chrobak et al. [20] schedule tasks in a multiprocessor environment to minimize maximum buffer size. Our algorithm does not add scheduling constraints to the problem: it reduces the total buffer size without affecting the schedule, and thereby not affecting the overall speed.

The techniques of Murthy et al. [21], [22], [23], [24], Teich et al. [25], and Geilen et al. [26] are closest to ours. They describe several algorithms for merging buffers in signal processing systems that use synchronous data flow models [27]. Govindarajan et al. [28] minimize buffer space while executing at the optimal computation rate in dataflow networks. They cast this as a linear programming problem. Sofronis et al. [29] 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 [30], [31] 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. In our previous work [2], we build a synchronous system to find deadlocks in a SHIM program. We 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.

There is a partial evaluation method [9] for SHIM that combines multiple concurrent processes to produce sequential code. Again, the work makes 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, in this paper, 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.

## XI. 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. To avoid state space explosion, we can treat the program as consisting of separate pieces.

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 and do not differentiate between scheduling choices and conditional branches. This may lead to false positives: our technique can discard pairs even though they 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. We found we could save 286KB of the PPE's memory for an FFT example and 983KB for a JPEG 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. 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 independently of the remaining sets.

We are now ignoring SHIM's exceptions [32]. Exceptions in SHIM provide a convenient way to terminate peer tasks and they are deterministic in behavior. We plan to consider exceptions in the future.

Currently, local variables and buffers do not interact with each other. This is because we make the assumption that the two need not necessarily reside in the same memory. However, for shared memory systems, we plan to explore the possibility of interaction by doing live variable analysis on local variables.

## Acknowledgment

This work was supported by NSF grant CNS–0720292.

## References

[1] S. A. Edwards and O. Tardieu, "SHIM: A deterministic model for heterogeneous embedded systems," in *Proceedings of the International Conference on Embedded Software (Emsoft)*, (Jersey City, New Jersey), pp. 37–44, Sept. 2005.

[2] N. Vasudevan and S. 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), pp. 49–58, June 2008.

[3] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing 74: Proceedings of IFIP Congress 74*, (Stockholm, Sweden), pp. 471–475, North-Holland, Aug. 1974.

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

[5] N. Vasudevan and S. A. Edwards, "Celling SHIM: Compiling deterministic concurrency to a heterogeneous multicore," in *Proceedings of the Symposium on Applied Computing (SAC)*, vol. III, (Honolulu, Hawaii), pp. 1626–1631, Mar. 2009.
[6] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, pp. 589–604, July/September 2005.
[7] M. Kistler, M. Perrone, and F. Petrini, "Cell multiprocessor communication network: Built for speed," *IEEE Micro*, vol. 26, pp. 10–23, May-June 2006.
[8] S. A. Edwards, N. Vasudevan, and O. Tardieu, "Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads," in *Proceedings of Design, Automation, and Test in Europe (DATE)*, (Munich, Germany), pp. 1498–1503, Mar. 2008.
[9] S. A. Edwards and O. Tardieu, "Efficient code generation from SHIM models," in *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, (Ottawa, Canada), pp. 125–134, June 2006.
[10] N. Vasudevan, S. Singh, and S. A. Edwards, "A deterministic multi-way rendezvous library for Haskell," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, (Miami, Florida), pp. 1–12, Apr. 2008.
[11] S. A. Edwards and O. Tardieu, "SHIM: A deterministic model for heterogeneous embedded systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, pp. 854–867, Aug. 2006.
[12] S. A. Edwards and J. Zeng, "Static elaboration of recursion for concurrent software," in *Proceedings of the Workshop on Partial Evaluation and Program Manipulation (PEPM)*, (San Francisco, California), pp. 71–80, Jan. 2008.
[13] E. Malaguti, M. Monaci, and P. Toth, "Models and heuristic algorithms for a weighted vertex coloring problem," *Journal of Heuristics*, 2008. DOI: 10.1007/s10732-008-9075-1.
[14] E. Malaguti, "The vertex coloring problem and its generalizations," *4OR: A Quarterly Journal of Operations Research*, vol. 7, pp. 101–104, Mar. 2009.
[15] N. Vasudevan and S. A. Edwards, "Buffer sharing in CSP-like programs," in *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, (Cambridge, Massachusetts), July 2009.
[16] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, "The design and implementation of a first-generation CELL processor," in *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, vol. 1, (San Francisco, California), pp. 184–185, 582, Feb. 2005.
[17] E. de Greef, F. Catthoor, and H. 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*, (Washington, DC, USA), p. 66, IEEE Computer Society, 1997.
[18] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Proceedings of the International Conference on Compiler Construction (CC)*, vol. 2304 of *Lecture Notes in Computer Science*, (Grenoble, France), pp. 179–196, Apr. 2002.
[19] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe, "Cache aware optimization of stream programs," in *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, (New York, NY, USA), pp. 115–126, ACM, 2005.
[20] M. Chrobak, J. Csirik, C. Imreh, J. Noga, J. Sgall, and G. J. Woeginger, "The buffer minimization problem for multiprocessor scheduling with conflicts," in *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, vol. 2076 of *Lecture Notes in Computer Science*, (Heraklion, Crete, Greece), pp. 862–874, Springer-Verlag, 2001.
[21] P. K. Murthy and S. S. Bhattacharyya, "Systematic consolidation of input and output buffers in synchronous dataflow specifications," *IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 673–682, 2000.
[22] P. K. Murthy and S. S. Bhattacharyya, "Shared buffer implementations of signal processing systems using lifetime analysis techniques," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 177–198, Feb. 2001.
[23] P. K. Murthy and S. S. Bhattacharyya, "Buffer merging—a powerful technique for reducing memory requirements of synchronous dataflow specifications," *ACM Transactions on Design Automation of Electronic Systems*, vol. 9, pp. 212–237, Apr. 2004.
[24] P. K. Murthy and S. S. Bhattacharyya, *Memory Management for Synthesis of DSP Software*. Boca Raton, FL, USA: CRC Press, Inc., 2006.
[25] J. Teich, E. Zitzler, and S. S. Bhattacharyya, "Buffer memory optimization in DSP applications—an evolutionary approach," in *Proceedings of Parallel Problem Solving from Nature (PPSN)*, vol. 1498 of *Lecture Notes in Computer Science*, (Amsterdam, The Netherlands), pp. 885–896, Springer-Verlag, Sept. 1998.
[26] M. Geilen, T. Basten, and S. Stuijk, "Minimising buffer requirements of synchronous dataflow graphs with model checking," in *Proceedings of the 42nd Design Automation Conference*, (Anaheim, California), pp. 819–824, ACM, 2005.
[27] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, pp. 1235–1245, Sept. 1987.
[28] R. Govindarajan, G. R. Gao, and P. D. Y, "Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks," *Journal of VLSI Signal Processing Systems*, vol. 31, pp. 207–209, July 2002.
[29] C. Sofronis, S. Tripakis, and P. Caspi, "A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling," in *Proceedings of the International Conference on Embedded Software (Emsoft)*, (New York, NY, USA), pp. 21–33, ACM, 2006.
[30] B. Lin, "Efficient compilation of process-based concurrent programs without run-time scheduling," in *Proceedings of Design, Automation, and Test in Europe (DATE)*, (Paris, France), pp. 211–217, Feb. 1998.
[31] B. Lin, "Software synthesis of process-based concurrent programs," in *Proceedings of the 35th Design Automation Conference*, (San Francisco, California), pp. 502–505, June 1998.
[32] O. Tardieu and S. A. Edwards, "Scheduling-independent threads and exceptions in SHIM," in *Proceedings of the International Conference on Embedded Software (Emsoft)*, (Seoul, Korea), pp. 142–151, Oct. 2006.

## RESPONSE TO REVIEWERS' COMMENTS

*Reviewer 1*

> *The main weakness of the paper is the restriction to SHIM, which limits the applicability of the approach. Most real applications are defined outside of the SHIM language (and probably would require serious efforts to be ported).*

While our work focuses on SHIM, we believe that the algorithm can be extended to other languages. We discuss how in the newly added Section IX.

> *Another weakness is that buffer sharing obviously only helps if the shared buffers are located on the same processing element.*

Yes, but our algorithm can also be used for distributing buffers on different processing elements. See Section VIII-B.

> *It would improve the article, if you could compare it with the base Memocode paper and outline any additions (Table II, III; Section VIII).*

We conducted more experiments (Table II and Table III) and reported them in Section VII. Also, we discuss how our algorithm can be applied in a distributed setting like the Cell Broadband Engine in Section VIII-A. In Section VIII-B, we provide a strategy for distributed buffer allocation. We discuss the limitations of our approach in Section IX. We also made small additions to Section I, Section III and Section XI.

> *page 1, end of Section I: mention also the (newly added) section VIII page 2, left column, first paragraph: "next" is used many times, with different meaning; consider rephrasing Listings Fig. 2,3,4 should be reformatted; the number of self-citations seems large*

We have made the changes. We replaced "next" by "send" and "recv." Hopefully, it is clearer now. We have also removed some of the self-citations.

*Reviewer 2*

> *What are exactly the lifetimes of the received channel values after a "next a" statement? And are these values kept in the buffer of the channel or is a copy operation performed?*

The sender writes (copies) data to the buffer. The receivers copy it from the buffer, after which data in the buffer is no longer used.

> *It looks like that every "next a" statement has to copy the value into a local variable. However, sharing buffer space with the local variable thus eliminating the need for the copy operation and the duplicate memory requirements of the local variable and channel buffer would interfere with the buffer sharing algorithm as presented. I would like a short discussion how channel buffers and memory for local variables interact.*

The buffer and the local variables need not necessarily reside in the same processing element. For instance, in Section VIII-A, we show how buffers can reside in the PPE while the local variables reside in the SPE. In our model, the local variables and the channel buffers do not interact. But if they do reside in the same memory, then we need additional analysis like liveness to see how they can share memory.

> *Nitpicks: Section VI its a "weighted vertex coloring" not "weighted vertex cover" problem. The references [15, 16] are OK again.*

Our mistake—we fixed this.

*Reviewer 3*

> *For a paper of this nature, the author should more clearly establish up-front what is general vs. SHIM-specific. Which aspects of the optimizations are generally beneficial to the compiler optimization community vs. what is limited to SHIM or CSP programs? Which aspects of the analysis are generally feasible to other languages vs. limited to specific properties of SHIM or CSP programs?*

We added Section IX to discuss this.

> *The paper is sprinkled with disclaimers about various simplifications to the analysis (for example, following both branches, replicating functions, limiting the stack depth during composition). They are important practical concerns when implementing the ideas in this paper, but as they are currently presented, they will be easy to miss. The authors only mention in passing that these simplifications do not impact the quality of the analysis/optimization. It would be useful for the authors to actually evaluate and report their impact quantitatively. For example, how many opportunities are you losing by assuming both sides of a branch are followed vs. an ideal analysis?*

There are a few programs like the one discussed in Section IX that do suffer from this. But, for practical the programs discussed in Section VII, we never lost any opportunity for sharing by assuming both sides of a branch are followed vs. an ideal analysis. Also, doing branch prediction at compile time is not feasible because the branch condition may well depend on program input.

> *The paper spends a lot of real-estate stating and proofing lemmas about the associativity and commutativity of automata composition. Is this necessary?*

The results of our lemmas are not surprising, but what we're doing isn't exactly classical automata composition because SHIM's rendezvous semantics are a little different. In any case, we wanted to include the complete proofs.

> *The discussion of Table IV in Section VII should do a much better job explaining the interesting data points. For example, why do Source-Sync and its modified pipelined version behave so differently? What does Framebuffer not get any savings?*

We fixed this. In summary, the pipeline uses larger buffers. The tasks in Framebuffer communicate through synchronization signals that hardly need any buffers.

> *Section VIII spends a full column describing a JPEG implementation in SHIM for the Cell processor. Besides staking a high-level claim, it doesn't add much in technical insights into the analysis or the optimization.*

Yes, but we felt that the paper would be incomplete without a discussion of a specific application for our algorithm.

> *The author could help the reviewers by identifying the difference between the current journal draft and the original conference draft*

The first reviewer also asked about this; see our response above.

**Nalini Vasudevan** received her B.E. degree in Computer Science from R.V. College of Engineering (Visweswariah Technological University), Bangalore, India in 2005, and her M.S. degree in Computer Science from Columbia University, New York in 2007.

She is currently a PhD student in the Department of Computer Science, Columbia University, New York. She has worked with Bell Labs, IBM TJ Watson Lab, Microsoft Research and Yahoo! in the past. Her research interests include programming principles, concurrency and verification.

**Stephen A. Edwards** received the B.S. degree in electrical engineering from the California Institute of Technology, Pasadena, in 1992, and the M.S. and Ph.D. degrees in electrical engineering from the University of California, Berkeley, in 1994 and 1997, respectively.

He is currently an Associate Professor with the Department of Computer Science, Columbia University, New York, NY, where he has been with since 2001 after a three-year stint with Synopsys, Inc., Mountain View, CA. His research interests include embedded system design, domain-specific languages, and compilers.