UNIVERSITY OF CAMBRIDGE

Deadlock Avoidance and Combinational Balancing for High-Level Synthesis

David Greaves
University of Cambridge
Computer Laboratory

David.Greaves@cl.cam.ac.uk

Abstract

The Bluespec and Kiwi tool chains project systems of communicating processes into hardware circuits. When a number of proceses are composed, two problems commonly arise at the system level: deadlock and excessive combinational delay. Both problems are emergent as the system grows and are best solved using a global pass of the whole assembly, rather than by systematic modification to components before composition.

Talk Overview

- Design expression using concurrent languages is encouraged.

- Syntax-driven, one action per clock cycle? Too crude, too many registers.

- We need automatic heuristic-based retiming/pipeline generation.

- This talk:
  Outline of an approach being implemented in Kiwi compiler but generally suitable. *NEED TO AOVID DEADLOCK.*

Join Calculus

```
//A simple join chord:
public class Buffer
{
  public async Put(char c);
  public char Get(bool f) & Put(char c) { return (f)?
toupper(c):c; }
}
```

Bluespec Verilog (BSV)

```
// Asimple rule
rule  rule1 (emptyflag && req);
    emptyflag <= false;
    ready <= true;
endrule
```

As we schedule more operations in one clock cycle, combinational delay in the guards builds up.

It is counter-intuitive to insert manual pipelining instructions at the appropriate granularity in such high-level source code.

Previous Work / State of Art ?

DATE 2010: "Automatic Pipelining from Transactional Datapath Specifications" By Nurvitadhi, Hoe, Kam and Lu.

- Automatic generation of scoreboards and forwarding but manual allocation of logic to pipeline stages.

Memocode 2011: "Controller Synthesis for Pipelined Circuits Using Uninterpreted Functions" by Georg Hofferek and Roderick Bloem:

- Builds BDDs for the system but then does not use these to check for liveness.

Bluespec System Verilog Compiler (BSV):

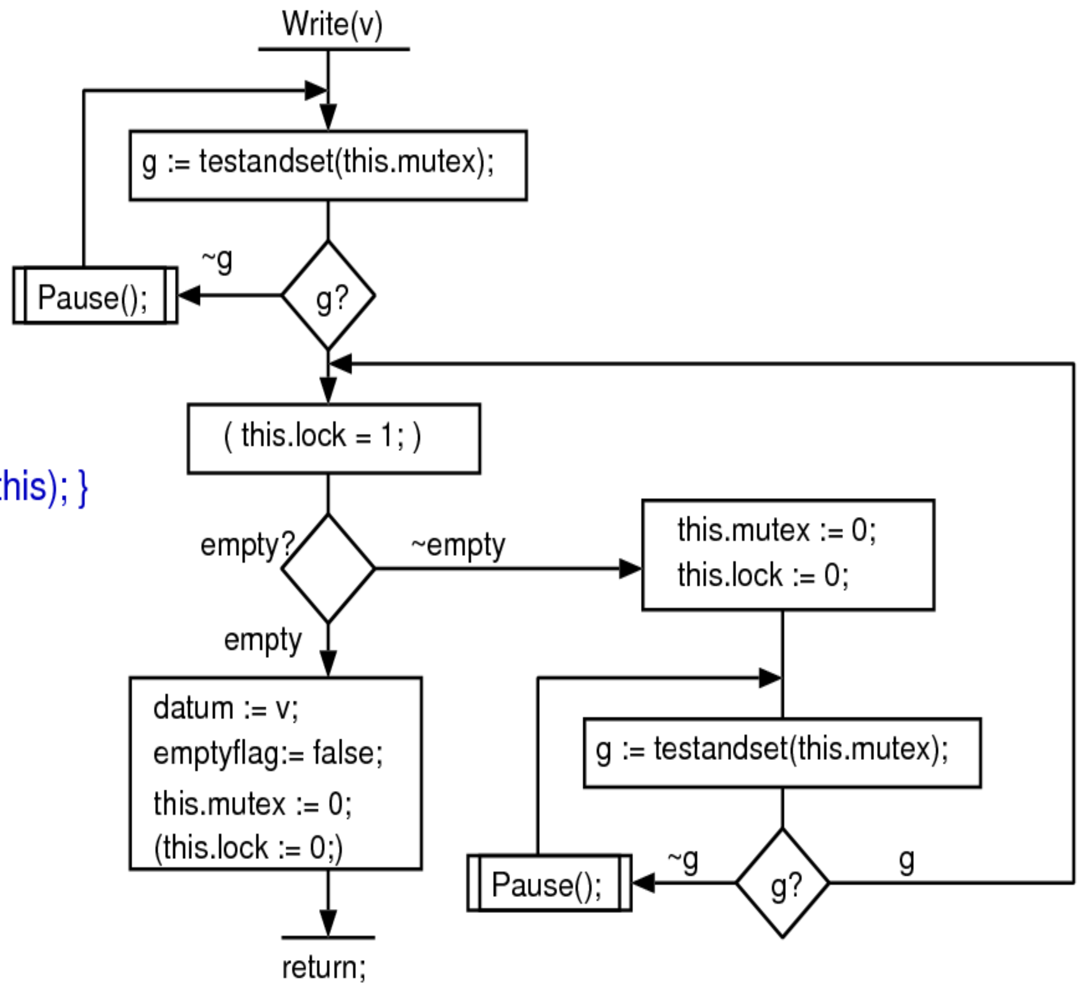
- Designer explicitly instantiates various FIFO stages with appropriate balance of combinational and sequential paths.

Kiwi HLS Approach

- Use the .net library concurrency primitives
- Below a certain level, replace implementations with our own hardware alternatives
- This is ultimately a shared-variable model with exclusion locks.
- Our implementation of one-place buffers gave poor performance compared with BSV implementation of the same structures.

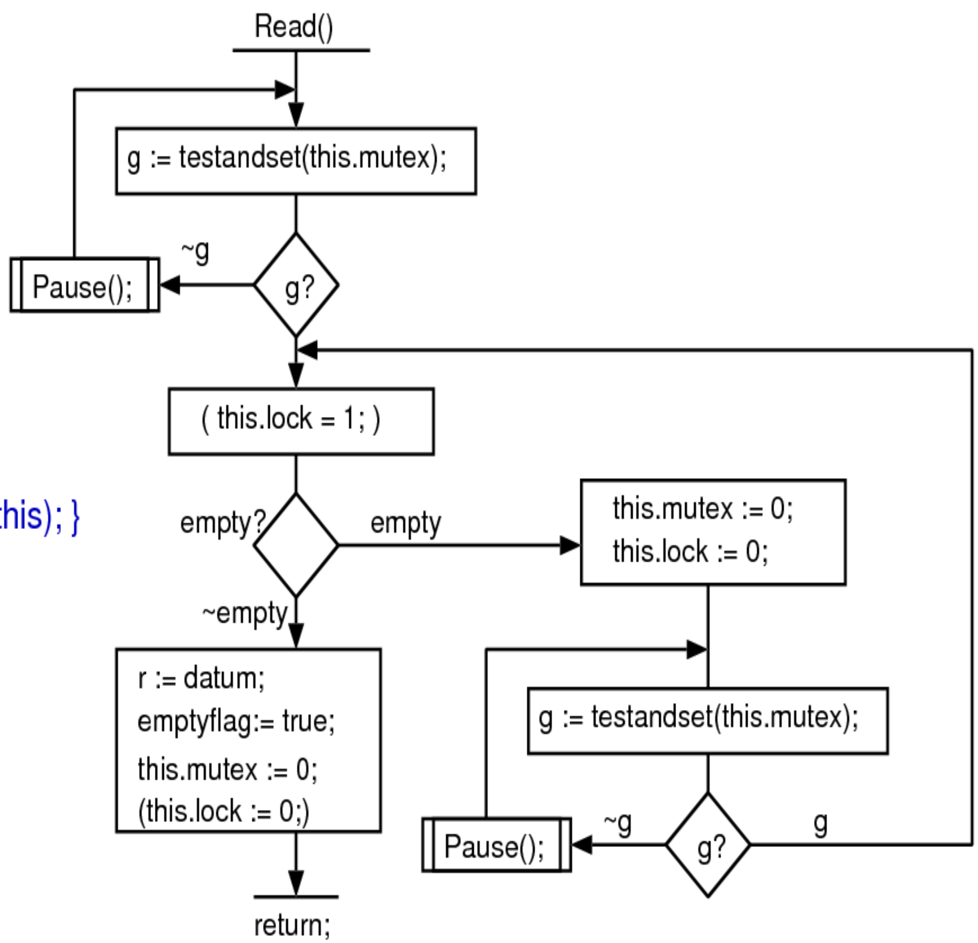
One-place buffer: Write Method

```
public class Channel<T>
  {
     T datum;
     volatile bool emptyflag = true;

     public void Write(T v)
     {  lock (this)
        {
          while (!emptyflag) {  Monitor.Wait(this); }
          datum = v;
          emptyflag = false;
          Monitor.PulseAll(this);
        }
     }
     ...
 }
```

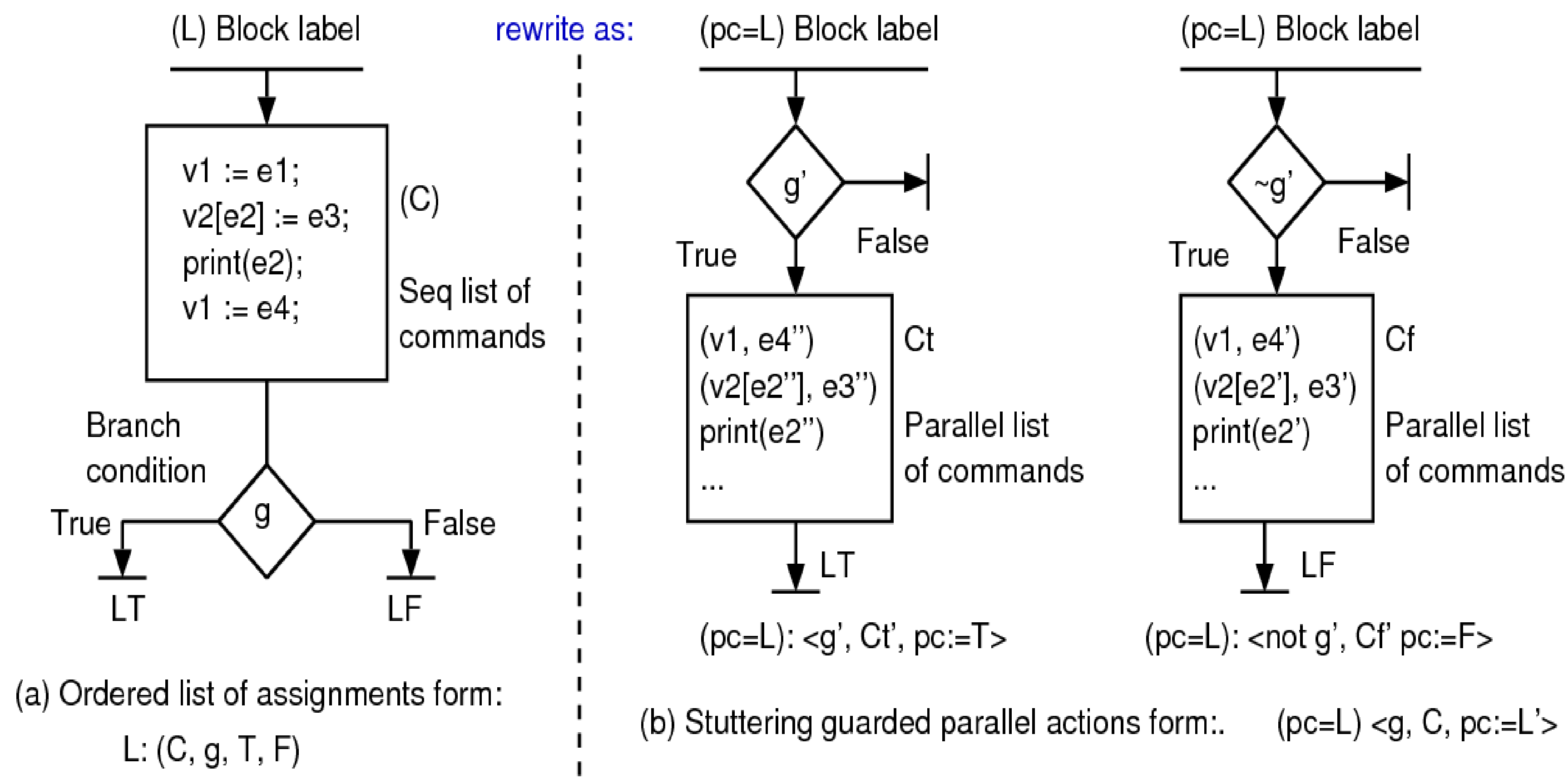


One-place buffer: Read Method

```
public class Channel<T>
  {
    T datum;
    volatile bool emptyflag = true;
    ...
    public T Read()
    {
      T r;
      lock (this)
      {  while (emptyflag)  {  Monitor.Wait(this); }
         emptyflag = true;
         r = datum;
         Monitor.PulseAll(this);
      }
      return r;
    }
 }
```

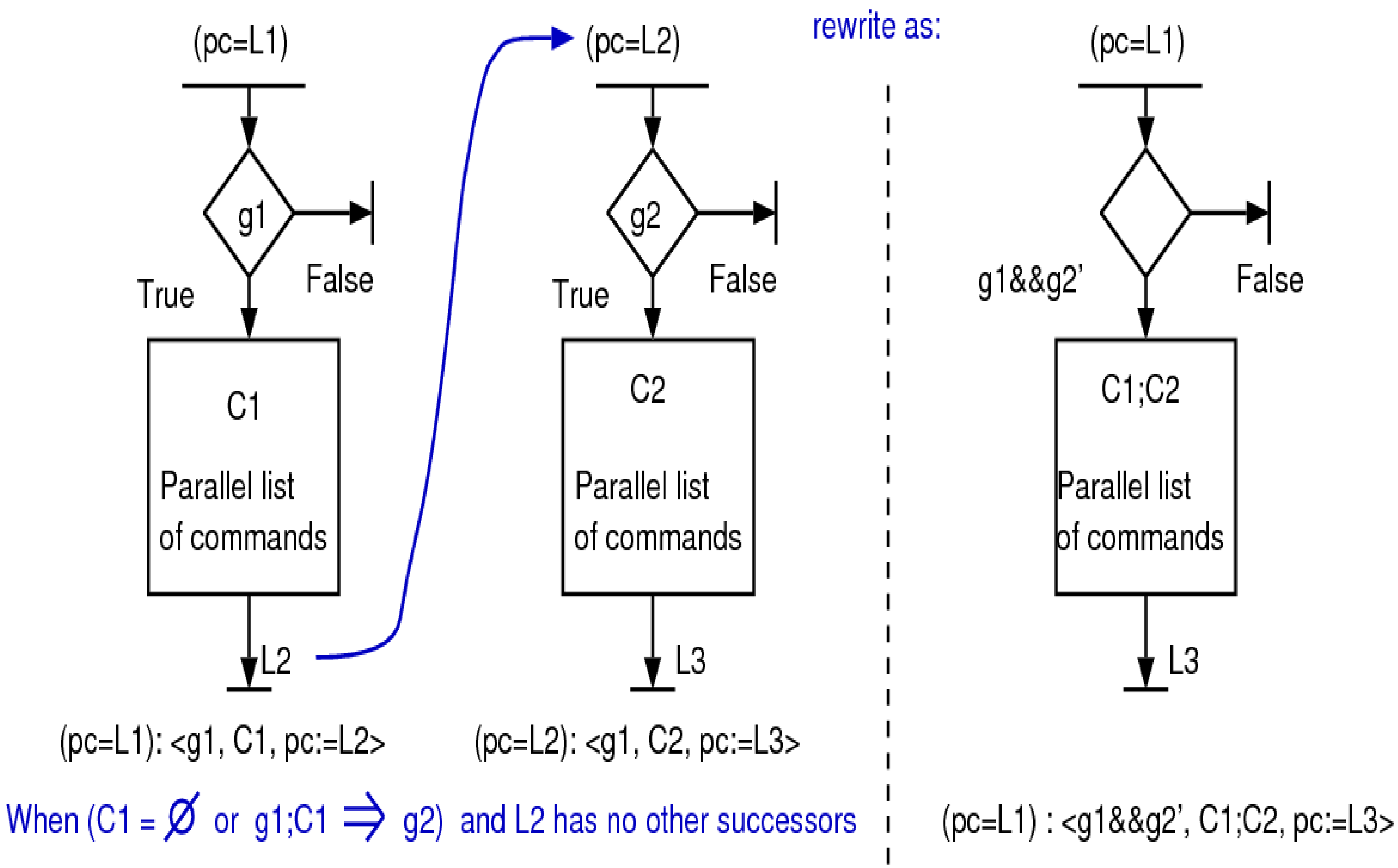


KiwiC compiler converts basic blocks in each thread to a sea of guarded actions.



Using symbolic elaboration we convert each path through a BB to a guarded action block. This gives an algebraic canonical form amenable to a host of rewrite rules.

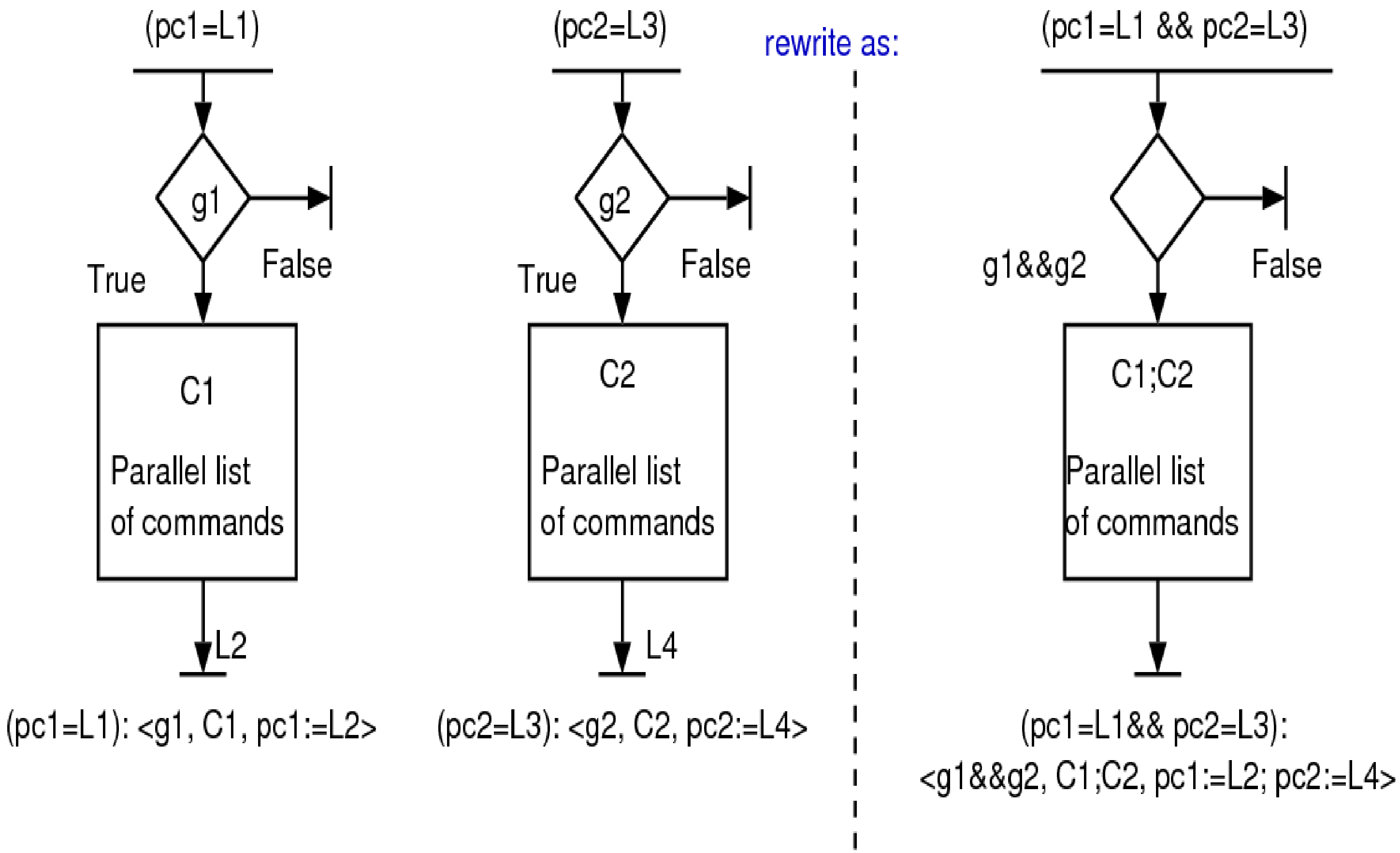
Guarded actions: Sequential Composition Rule



There is a vast theory in the literature for composing and decomposing such rules.

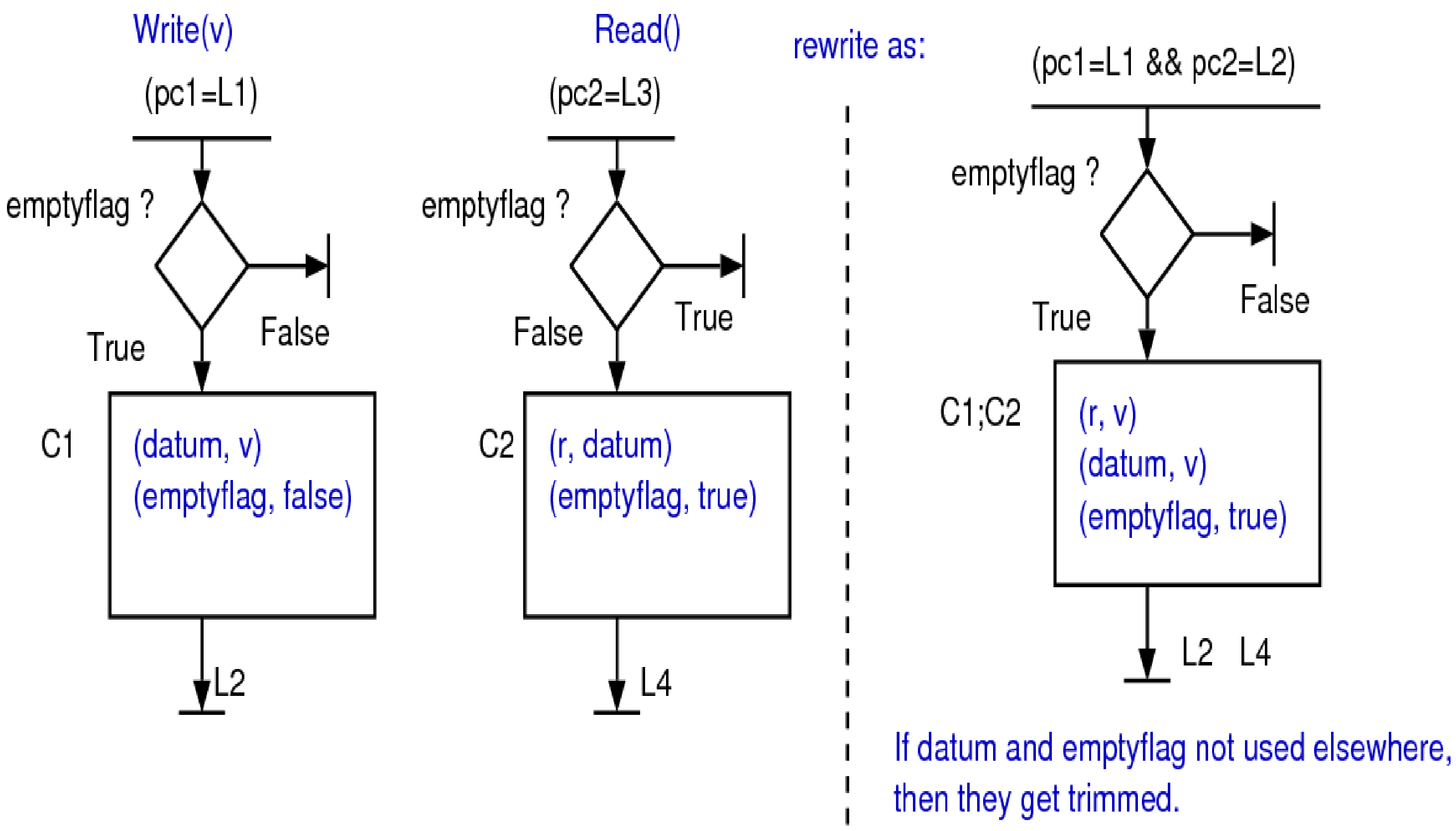
Guarded actions: Parallel Composition

Here we have forced a rendezvous between two threads.



Rendezvous can lead to deadlock and increases fan-in in guard conjunction. State space is reduced, especially sometimes.

Sequential Compose Write(v)|| Read()



If we compose in the other order, emptyflag is not left at its reset value and so does not globally disappear.

Design Space Search: Guiding Heuristic

When to compose : use A* or other search algorithm.

Need figure of merit for each trial based on :

- Composition eliminates registers totalling n bits $\rightarrow$ f(n)
  But might need to eliminate all occurrences
  to get the benefit?
- Balances seq/comb logic ratios: espresso-based logic depths are calculated on the fly for each trial composition.
- Decreases deadlock chances: later lock, earlier release? Do a mini model check after each trial?
- Reduces system latency.

Deadlock from Enforced Synchonicity

```
class PATHOLOCK2
{
    [Kiwi.OutputWordPort("fresult")]
    public static uint fresult;

    static Kiwi.Channel<uint> chan1a;
    static Kiwi.Channel<uint> chan1b;
    static Kiwi.Channel<uint> chan2;
```

Pathological example:
If we loose the asynchronous buffer within chan1b we cannot make our first write to chan1a.

```
public static void Producer()
{
   for (uint i = 100; i < 1000; i+=100)
    {   chan1b.Write(i+2); // Write b
        chan1a.Write(i+4); // before a
    }
}

public static void Stage()
{
    while (true)
    {   uint i = chan1a.Read();
        uint j = chan1b.Read();
        chan2.Write(100U + i + j);
    }
}
```

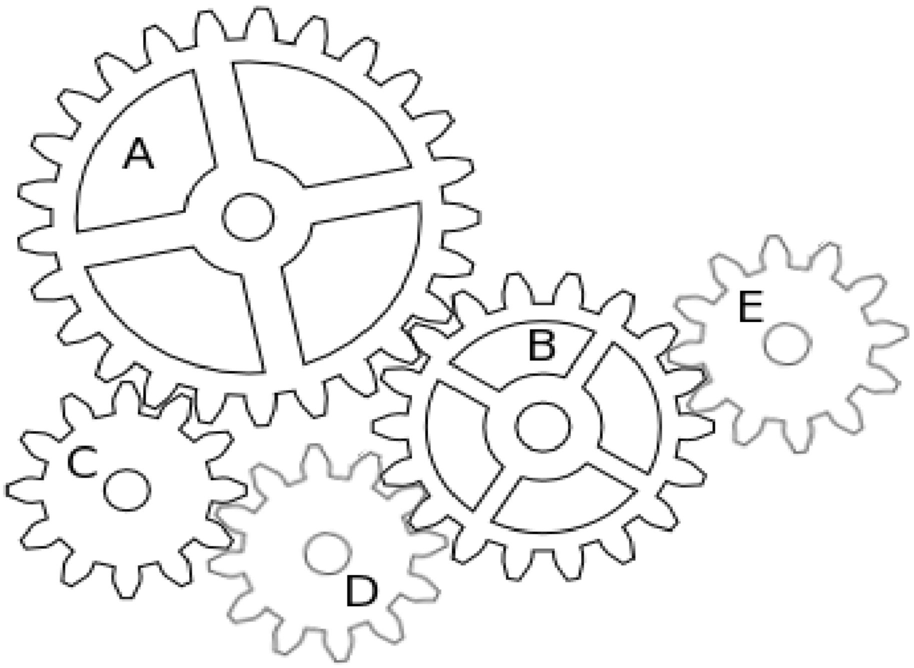
Main Program (is boring)

```
[Kiwi.HardwareEntryPoint()]
public static void Behaviour()
{
    chan1a = new Kiwi.Channel<uint>();
    chan1b = new Kiwi.Channel<uint>();
    chan2 = new Kiwi.Channel<uint>();

    Thread ProducerThread = new Thread(new ThreadStart (Producer));
    ProducerThread.Start();

    Thread StageThread = new Thread(new ThreadStart(Stage));
    StageThread.Start();

    while (true)
    {
        fresult = chan2.Read();
        Console.WriteLine("Result is " + fresult);
    }
  }
}
```

Output:

```
Result is 306
Result is 506
Result is 706
Result is 906
Result is 1106
Result is 1306
Result is 1506
Result is 1706
Result is 1906
```

Can it be that hard ?



Consider a system of cogs:

“If we insert cog wheel D the system CLEARLY deadlocks.”

Our threads are like cogs made of more bendy material.

Do we need a full-blooded modelcheck on every trial ?

Model check after every trial compositon?

- Deadlock check only the SCCs ? Often there aren't any...
- Integrated BDD-based symbolic model checker? Variable order finding... slow … slow …
- Aggressive partial order reduction (stubborn sets and dynamic POR?) … maybe ...
- Use Attie + Chockler conservative algorithms ?

Attie and Chockler's Master Stroke

“Efficiently verifiable conditions for deadlock-freedom of large concurrent programs” Paul C. Attie, Hana Chockler (Boston).

In VMCAI'05 Proceedings of the 6th international conference on Verification, Model Checking, and Abstract Interpretation.

> We present two polynomial-time algorithms for automatic verification of deadlock-freedom of large finite-state concurrent programs. We consider shared-memory concurrent programs in which a process can nondeterministically choose amongst several (enabled) actions at any step...

A generalisation of the standard AND/OR knot finding approach in the wait for graph (WFG) suitable for general shared-variable action systems.

Build a bi-partite graph relating the edges to the actions.

Only need to check interactions of three machines at a time to determine complete system's deadlock freeness: polynomial space and time w.r.t. all metrics.

Conclusion

- Concurrent expression of design intent is good: plenty of parallelism available.
- Syntax-directed or manual expression of pipeline stages is inflexible/infelicitious.
- Automatic balancing while avoiding dead-lock looks totally feasible since brutal mod-el-checking can be avoided.
- New static schedulers for concurrent spe-cification languages shall emerge ...