

SHIM: A Deterministic Concurrent Language

Stephen A. Edwards

Columbia University


April 2009

SHIM



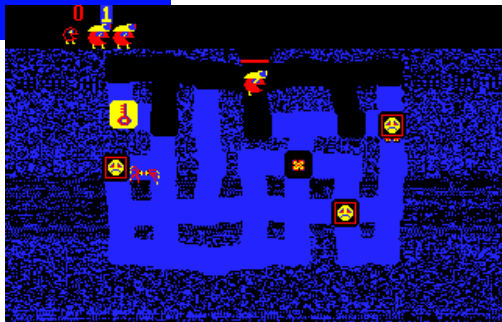
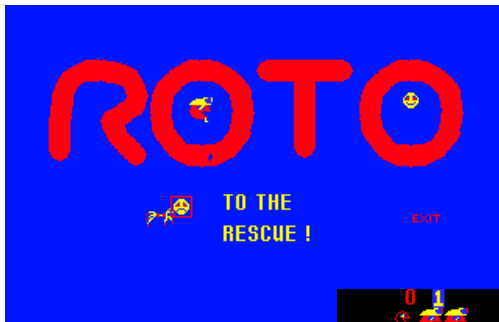
shim \ 'shim \ *n*

1 : a thin often tapered piece of material (as wood, metal, or stone) used to fill in space between things (as for support, leveling, or adjustment of fit).



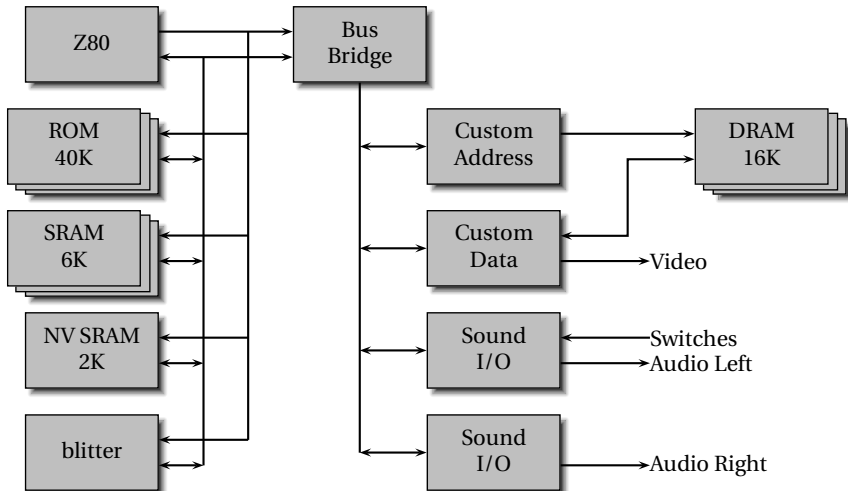
2 : *Software/Hardware Integration Medium*, a model for describing hardware/software systems

Robby Roto



(Bally/Midway 1981)

Robby Roto Block Diagram





- *Concurrent*
Hardware always concurrent
- *Mixes synchronous and asynchronous styles*
Need multi-rate for hardware/software systems
- *Only requires bounded resources*
Hardware resources fundamentally bounded
- *Formal semantics*
Do not want arguments about what something means
- *Scheduling-independent*
Want the functionality of a program to be definitive
Always want simulated behavior to reflect reality
Verify functionality and performance separately

An imperative language with familiar C/Java-like syntax

```
int32 gcd(int32 a, int32 b)
{
  while (a != b) {
    if (a > b)
      a -= b;
    else
      b -= a;
  }
  return a;
}
```

```
struct foo { // Composite types
  int x;
  bool y;
  uint15 z; // Explicit-width integers
  int<-3,5> w; // Explicit-range integers
  int8 p[10]; // Arrays
  bar q; // Recursive types
};
```

Three Additional Constructs

`stmt1 par stmt2`

Run `stmt1` and `stmt2` concurrently

`send var`

Communicate on channel `var`

`recv var`

`next var`

`try {`

Define the scope of an exception

`⋮`

`throw exc`

Raise an exception

`⋮`

`} catch(exc) stmt`

Concurrency & *par*

Par statements run concurrently and asynchronously

Terminate when all terminate

Each thread gets private copies of variables; no sharing

Writing thread sets the variable's final value

```
void main() {  
  int a = 3, b = 7, c = 1;  
  {  
    a = a + c; // a ← 4, b = 7, c = 1  
    a = a + b; // a ← 11, b = 7, c = 1  
  } par {  
    b = b - c; // a = 3, b ← 6, c = 1  
    b = b + a; // a = 3, b ← 9, c = 1  
  }  
  // a ← 11, b ← 9, c = 1  
}
```

Restrictions

Both pass-by-reference and pass-by-value arguments

Simple syntactic rules avoid races

```
void f(int &x) { x = 1; } // x passed by reference
```

```
void g(int x) { x = 2; } // x passed by value
```

```
void main() {
```

```
  int a = 0, b = 0;
```

```
  a = 1; par b = a; // OK: a and b modified separately
```

```
  a = 1; par a = 2; // Error: a modified by both
```

```
  f(a); par f(b); // OK: a and b modified separately
```

```
  f(a); par g(a); // OK: a modified by f only
```

```
  g(a); par g(a); // OK: a not modified
```

```
  f(a); par f(a); // Error: a passed by reference twice
```

```
}
```

Communication

Blocking: thread waits for all processes that know about a

```
void f(chan int a) { // a is a copy of c
  a = 3; // change local copy
  recv a; // receive (wait for g)
           // a now 5
}

void g(chan int &b) { // b is an alias of c
  next b = 5; // sets c and send (wait for f)
              // b now 5
}

void main() {
  chan int c = 0;
  f(c); par g(c);
}
```

Synchronization, Deadlocks

Blocking communication makes for potential deadlock

```
{ next a; next b; } par { next b; next a; } // deadlocks
```

Only threads responsible for a variable must synchronize

```
{ next a; next b; } par next b; par next a; // OK
```

When a thread terminates, it is no longer responsible

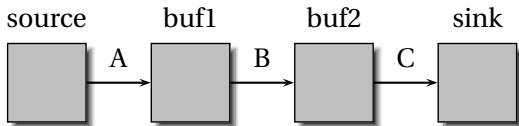
```
{ next a; next a; } par next a; // OK
```

Philosophy: deadlocks easy to detect; races are too subtle

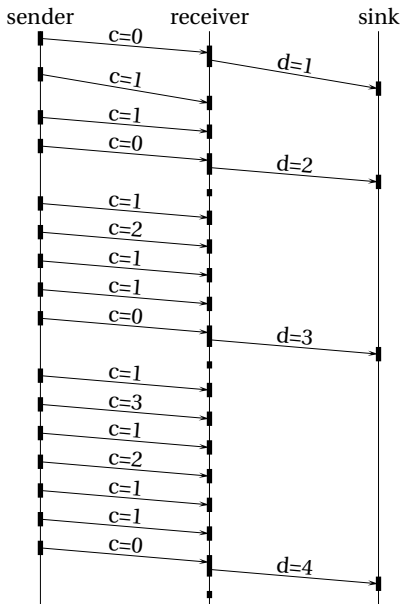
SHIM prefers deadlocks to races (always reproducible)

An Example

```
void main() {  
  chan uint8 A, B, C;  
  { // source: generate four values  
    next A = 17;  
    next A = 42;  
    next A = 157;  
    next A = 8;  
  } par { // buf1: copy from input to output  
    for (;;) {  
      next B = next A;  
    }  
  } par { // buf2: copy, add 1 alternately  
    for (;;) {  
      next C = next B;  
      next C = next B + 1;  
    }  
  } par { // sink  
    for (;;) {  
      recv C;  
    }  
  }  
}
```



Message Sequence Chart



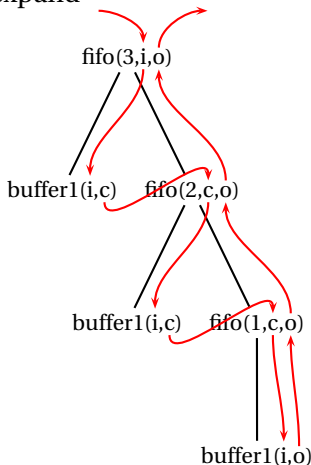
```
int a, b; chan int c, d;
{
  d = 0;
  for (;;) {
    while (e > 0) {
      next c = 1;
      next c = e;
      e = e - 1;
    }
    next c = 0;
    next d = d + 1;
  }
}
par {
  a = b = 0;
  for (;;) {
    do {
      if (next c != 0)
        a = a + next c;
    } while (c);
    b = b + 1;
  }
}
par {
  for (;;) recv d;
}
```

Recursion & Concurrency

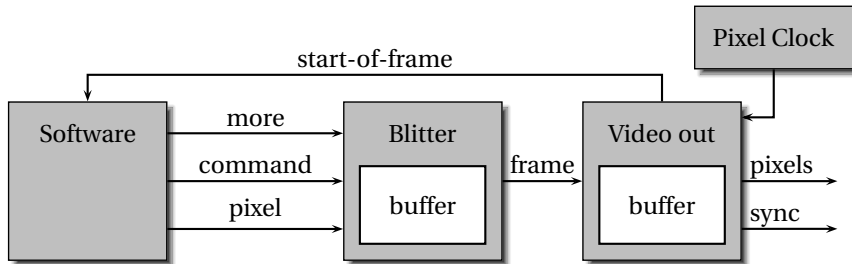
A bounded FIFO: compiler will analyze & expand

```
void buffer1(chan int in, chan int &out) {  
    for (;;) next out = next in;  
}
```

```
void fifo(int n, chan int in, chan int &out) {  
    if (n == 1)  
        buffer1(in, out);  
    else {  
        chan int channel;  
        buffer1(in, channel);  
        par  
            fifo(n-1, channel, out);  
    }  
}
```



Robby Roto in SHIM



```
while (player is alive) {  
  next start-of-frame;  
  ... game logic...  
  next more = true;  
  next command = ...;  
  ... game logic...  
  next more = false;  
}  
  
  for (;) {  
    while (next more) {  
      next command;  
      Write to buffer  
    }  
    next frame = buffer;  
  }  
  
  for (;) {  
    next start-of-frame;  
    for each line {  
      next sync = ...;  
      for each pixel {  
        next clock;  
        Read pixel  
        next pixel = ...;  
      }  
    }  
    buffer = next frame;  
  }
```

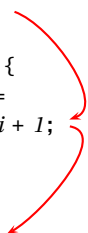
Exceptions

Sequential semantics are classical

```
void main() {  
    int i = 1;  
    try {  
        throw T;  
        i = i * 2; // Not executed  
    } catch (T) {  
        i = i * 3; // Executed by throw T  
    }  
        // i = 3 on exit  
}
```

Exceptions & Concurrency

```
void main() {  
  chan int i = 0, j = 0;  
  try {  
    while (i < 5)  
      next i = i + 1;  
    throw T;  
  } par {  
    for (;;) {  
      next j =  
        next i + 1;  
    }  
  } par {  
    for (;;)  
      recv j;  
  } catch (T) {}  
}
```



Exceptions propagate through communication actions to preserve determinism

Idea: “transitive poisoning”

Raising an exception “poisons” a process

Any process attempting to communicate with a poisoned process is itself poisoned (within exception scope)

“Best effort preemption”

Another Example

Five functions that call each other and communicate through channel *A*

```
void main() {  
  try {  
    chan int A;  
    f(A); par g(A);  
  } catch (Done) {}  
}
```

```
void f(chan int &A) throws Done {  
  h(A); par j(A);  
}
```

```
void g(chan int A) {  
  recv A;  
  recv A;  
}
```

```
void h(chan int &A) {  
  A = 4; send A;  
  A = 2; send A;  
}
```

```
void j(chan int A) throws Done {  
  recv A;  
  throw Done;  
}
```

Another Example

Parents call children

```
void main() {  
  try {  
    chan int A;  
    f(A); par g(A);  
  } catch (Done) {}  
}
```

```
void f(chan int &A) throws Done {  
  h(A); par j(A);  
}
```

```
void g(chan int A) {  
  recv A;  
  recv A;  
}
```

```
void h(chan int &A) {  
  A = 4; send A;  
  A = 2; send A;  
}
```

```
void j(chan int A) throws Done {  
  recv A;  
  throw Done;  
}
```

Another Example

h sends 4 on *A*,
g and *j* rendezvous

```
void main() {  
  try {  
    chan int A;  
    f(A); par g(A);  
  } catch (Done) {}  
}
```

```
void f(chan int &A) throws Done {  
  h(A); par j(A);  
}
```

```
void g(chan int A) {  
  recv A;  
  recv A;  
}
```

```
void h(chan int &A) {  
  A = 4; send A;  
  A = 2; send A;  
}
```

```
void j(chan int A) throws Done {  
  recv A;  
  throw Done;  
}
```

Another Example

j throws an exception. *g* and *h* poisoned by attempting communication

```
void main() {  
  try {  
    chan int A;  
    f(A); par g(A);  
  } catch (Done) {}  
}
```

```
void f(chan int &A) throws Done {  
  h(A); par j(A);  
}
```

```
void g(chan int A) {  
  recv A;  
  recv A;  
}
```

```
void h(chan int &A) {  
  A = 4; send A;  
  A = 2; send A;  
}
```

```
void j(chan int A) throws Done {  
  recv A;  
  throw Done;  
}
```

Another Example

Concurrent processes
terminate, control passed to
exception handler

```
void main() {  
  try {  
    chan int A;  
    f(A); par g(A);  
  } catch (Done) {}  
}
```

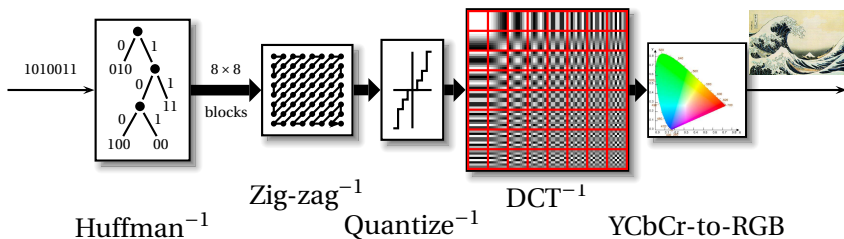
```
void f(chan int &A) throws Done {  
  h(A); par j(A);  
}
```

```
void g(chan int A) {  
  recv A;  
  recv A;  
}
```

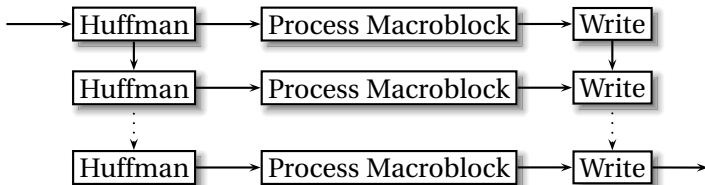
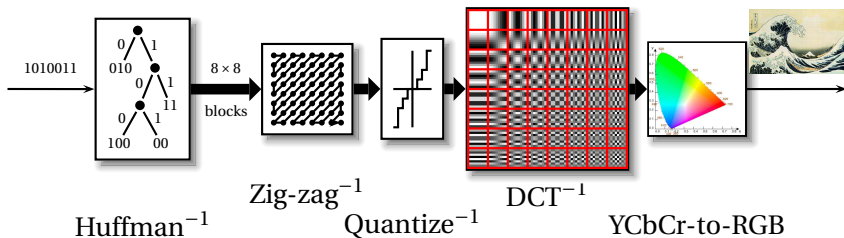
```
void h(chan int &A) {  
  A = 4; send A;  
  A = 2; send A;  
}
```

```
void j(chan int A) throws Done {  
  recv A;  
  throw Done;  
}
```

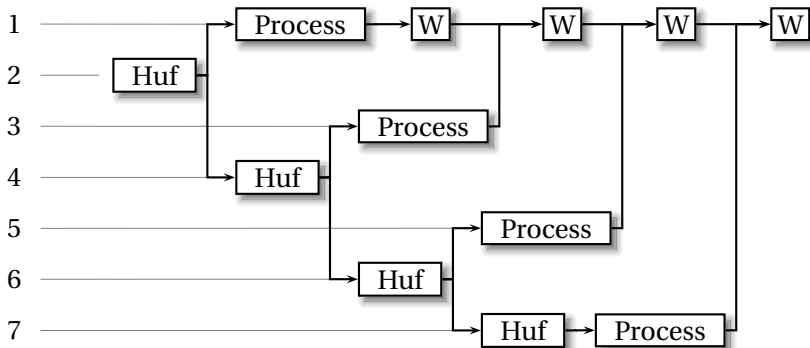
JPEG Decoding



JPEG Decoding



Seven-task JPEG schedule



Idea: minimize communication events

SHIM for the Seven-task Schedule

```
unpacker_state ustate;  
writer_state wstate;
```

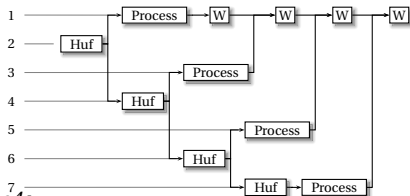
```
stripe stripe1, stripe2, stripe3, stripe4;
```

```
pixels pixels1; // to writer  
chan pixels pixels2, pixels3, pixels4;
```

```
void unpack(unpacker_state &state, stripe &stripe) { ... } // Huffman Deco
```

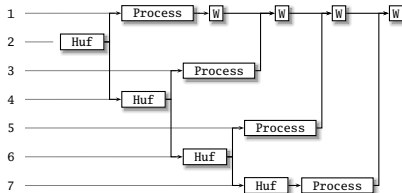
```
void process(const stripe &stripe, pixels &pixels) { ... } // IDCT, etc.
```

```
void write(writer_state &wstate, const pixels &pixels) { ... } // Write to file
```



SHIM for the Seven-task Schedule

```
unpack(ustate, stripe1); // 2
{
  process(stripe1, pixels1); write(wstate, pixels1); // 1
  recv pixels2; write(wstate, pixels2);
  recv pixels3; write(wstate, pixels3);
  recv pixels4; write(wstate, pixels4);
} par {
  unpack(ustate, stripe2); // 4
  {
    process(stripe2, pixels2); send pixels2; // 3
  } par {
    unpack(ustate, stripe3); // 6
    {
      process(stripe3, pixels3); send pixels3; // 5
    } par {
      unpack(ustate, stripe4); // 7
      process(stripe4, pixels4); send pixels4;
    } } }
} } }
```

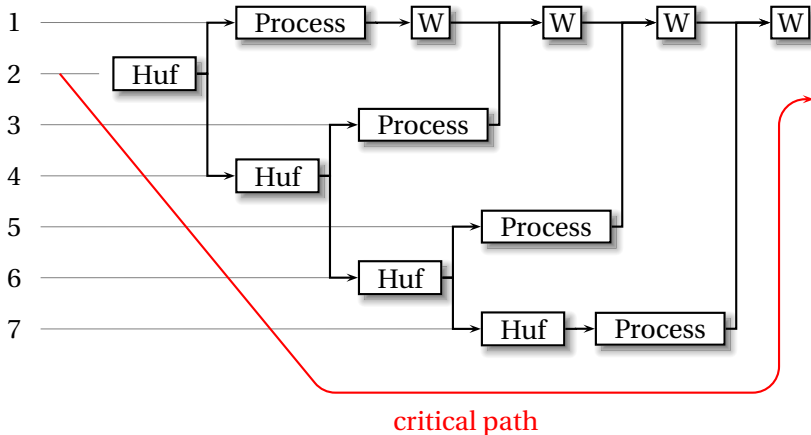


SHIM Enforces Dependencies

```
unpack(ustate, stripe1);
{
  process(stripe1, pixels1); write(wstate, pixels1);
  recv pixels2; write(wstate, pixels2);
  recv pixels3; write(wstate, pixels3);
  recv pixels4; write(wstate, pixels4);
} par {
  unpack(ustate, stripe2);
  {
    process(stripe2, pixels2); send pixels2;
  } par {
    unpack(ustate, stripe3);
    {
      process(stripe3, pixels3); send pixels3;
    } par {
      unpack(ustate, stripe4);
      process(stripe4, pixels4); send pixels4;
    }
  }
}
```

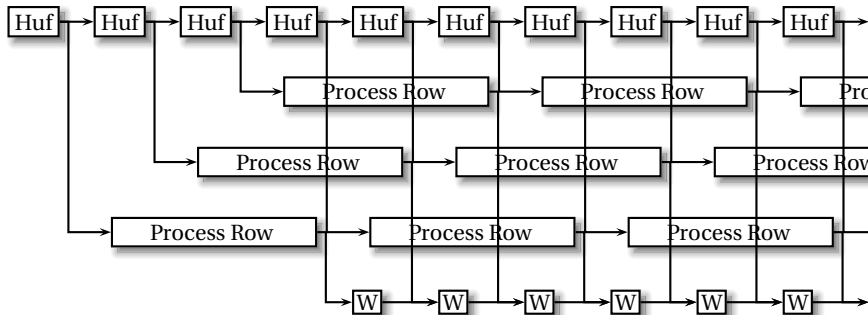
- Writer state local to one process
- Unpacker state can only be passed by reference once
- Trying to run *unpack* or *write* in parallel gives compiler error

Oops



Only achieved a 1.8× speedup

Pipelined JPEG

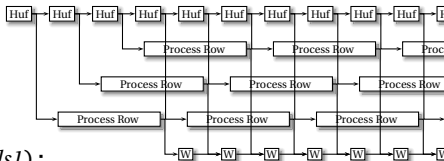


Process a row of blocks at a time (e.g., 64).

Reduce communication; accelerate start-up and termination.

SHIM for Pipelined JPEG

```
try {  
  {  
    for (;;) {  
      unpack(ustate, row1); send row1; if (--rows == 0) break;  
      unpack(ustate, row2); send row2; if (--rows == 0) break;  
      unpack(ustate, row3); send row3; if (--rows == 0) break;  
    } throw Done;  
  } par  
  process(row1, pixels1); par  
  process(row2, pixels2); par  
  process(row3, pixels3); par  
  {  
    for (;;) {  
      rcv pixels1; write(wstate, pixels1);  
      rcv pixels2; write(wstate, pixels2);  
      rcv pixels3; write(wstate, pixels3);  
    } }  
} catch (Done) {}
```







Pthreads Compiler: Task and Channel Structures

```
void foo(int a, int a)  
{  
  chan int c;  
}
```

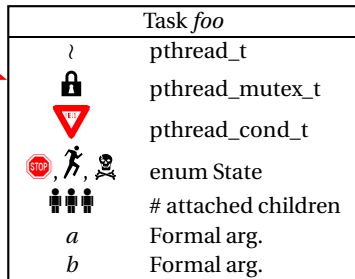
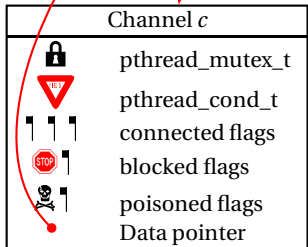
Pthreads Compiler: Task and Channel Structures

```
void foo(int a, int a)  
{  
  chan int c;  
}
```

Task <i>foo</i>	
}	pthread_t
	pthread_mutex_t
	pthread_cond_t
	enum State
	# attached children
<i>a</i>	Formal arg.
<i>b</i>	Formal arg.

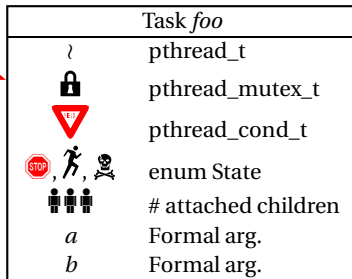
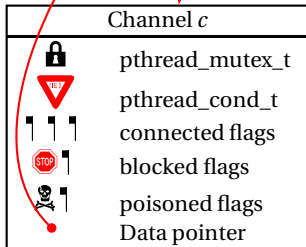
Pthreads Compiler: Task and Channel Structures

```
void foo(int a, int a)
{
  chan int c;
}
```



Pthreads Compiler: Task and Channel Structures

```
void foo(int a, int a)  
{  
  chan int c;  
}
```



```
void event_c() {  
  if (c.connected == c.blocked) {  
    // Communicate  
  } else if (c.poisoned) {  
    // Propagate exceptions  
  }  
}
```

Code for *send* A in h0

```
pthread_mutex_lock(A.mutex); // Lock for channel A
```

```
A.blocked |= (A_h|A_f|A_main);
```

```
// Block ancestors, too.
```

```
event_A(); // Communicate if possible
```

```
while (A.blocked & A_h) { // Are we ready?
```

```
  if (A.poisoned & A_h) { // Were we poisoned?
```

```
    pthread_mutex_unlock(A.mutex);
```

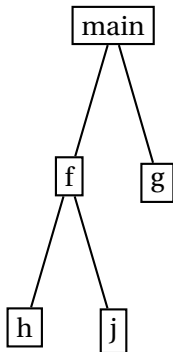
```
    goto _poisoned; // Handle exception
```

```
  }
```

```
  pthread_cond_wait(A.cond, A.mutex); // Yield
```

```
}
```

```
pthread_mutex_unlock(A.mutex);
```



An Event Function

```
void event_A0 {  
    unsigned int can_die = 0, kill = 0;  
    if(A.connected == A.blocked) {
```

```
// Flags  
// Communicate
```

```
} else if(A.poisoned) {
```

```
// Propagate exceptions
```

```
}}}
```

An Event Function

```
void event_A0 {  
    unsigned int can_die = 0, kill = 0;                                     // Flags  
    if(A.connected == A.blocked) {                                       // Communicate  
  
        A.blocked = 0;                                                    // Unblock everybody  
        if(A.connected & A_g) *A.g = *A.main;                             // Copy data  
        if(A.connected & A_j) *A.j = *A.main;                             // Copy data  
        pthread_cond_broadcast(A.cond);                                    // Awaken blocked tasks  
  
    } else if(A.poisoned) {                                              // Propagate exceptions  
  
        can_die = blocked & (A_g|A_h|A_j);                                // Compute can_die  
        if(can_die & (A_h|A_j) == A.connected & (A_h|A_j)) can_die |= blocked & A_f;  
        if(A.poisoned & (A_f|A_g)) {                                      // Compute kill  
            kill |= A_g; if(can_die & A_f) kill |= (A_f|A_h|A_j);  
        }  
        if(A.poisoned & (A_h|A_j)) { kill |= A_h; kill |= A_j; }  
        if(kill &= can_die & ~A.poisoned) {                               // Anybody to poison?  
            pthread_mutex_unlock(A.mutex);                                // Poison g if necessary  
            if(kill & A_g) {  
                pthread_mutex_lock(g.mutex);  
                g.state = POISON;  
                pthread_mutex_unlock(g.mutex); }  
            // also poison f, h, and j if in kill set...  
            pthread_mutex_lock(A.mutex);  
            A.poisoned |= kill; pthread_Cond_broadcast(A.cond);  
  
        }  
    }  
}
```

JPEG Experiment

21600 × 10800 .jpg file from NASA

Four-core Intel Xeon E5310

Sequential reference C code: .jpg to Sun rasterfile

Used the “pipelined” schedule

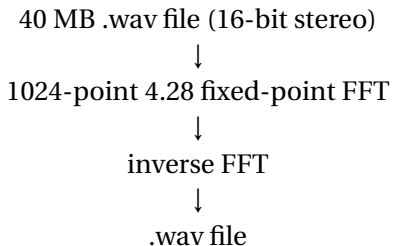
Measured speedup of 1–4 cores

Measured speedup of 1–5 IDCT tasks

JPEG Results

Cores	Tasks	Time	Total	Total/Time	Speedup
1	1	25s	20s	0.8	1.0×(def)
1	1+3+1	24	24	1.0	1.04
2	1+3+1	13	24	1.8	1.9
3	1+3+1	11	24	2.2	2.3
4	1+3+1	8.7	25	2.9	2.9
4	1+1+1	16	24	1.5	1.6
4	1+2+1	9.3	25	2.7	2.7
4	1+3+1	8.7	25	2.9	2.9
4	1+4+1	8.2	25	3.05	3.05
4	1+5+1	8.6	25	2.9	2.9

FFT Experiment (testing roundoff)



- Same hardware as JPEG (Xeon Quad-core)
- Baseline: sequential C from *Numerical Recipes*
- 1–4 cores, “pipelined” with 1 1024-sample block
- 1–4 cores, “pipelined” with 16 1024-sample blocks

FFT Results

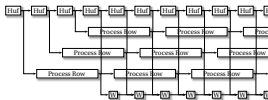
Code	Cores	Time	Total	Total/Time	Speedup
Handwritten C	1	2.0s	2.0s	1.0	1.0×(def)
Sequential SHIM	1	2.1	2.1	1.0	0.95
Parallel SHIM	1	2.1	2.1	1.0	0.95
Parallel SHIM	2	1.3	2.0	1.5	1.5
Parallel SHIM	3	0.92	2.1	2.2	2.2
Parallel SHIM	4	0.86	2.1	2.4	2.3
Parallel 16	1	1.9	1.9	1.0	1.1
Parallel 16	2	1.0	1.9	1.9	2.0
Parallel 16	3	0.88	1.9	2.1	2.2
Parallel 16	4	0.6	1.9	3.2	3.3

Conclusions

- Scheduling-independent message passing language

SHIM

- Exploring schedules interesting, safe



- Our compiler generates C code with pthreads calls



- Efficient: 3.05 and 3.3× speedups on a four-core

