Celling SHIM: Compiling Deterministic Concurrency to a Heterogeneous Multicore

Nalini Vasudevan and Stephen A. Edwards

Columbia University in the City of New York, USA

March 2009

SHIM

Main Points

- Scheduling-independent message passing works for parallel programming
  We use the SHIM language
- This paradigm helps to safely explore schedules
  Compiler catches race-related bugs
- Our compiler generates code that runs on the IBM CELL
  Synthesizing communication the trick

A SHIM 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);
}
```

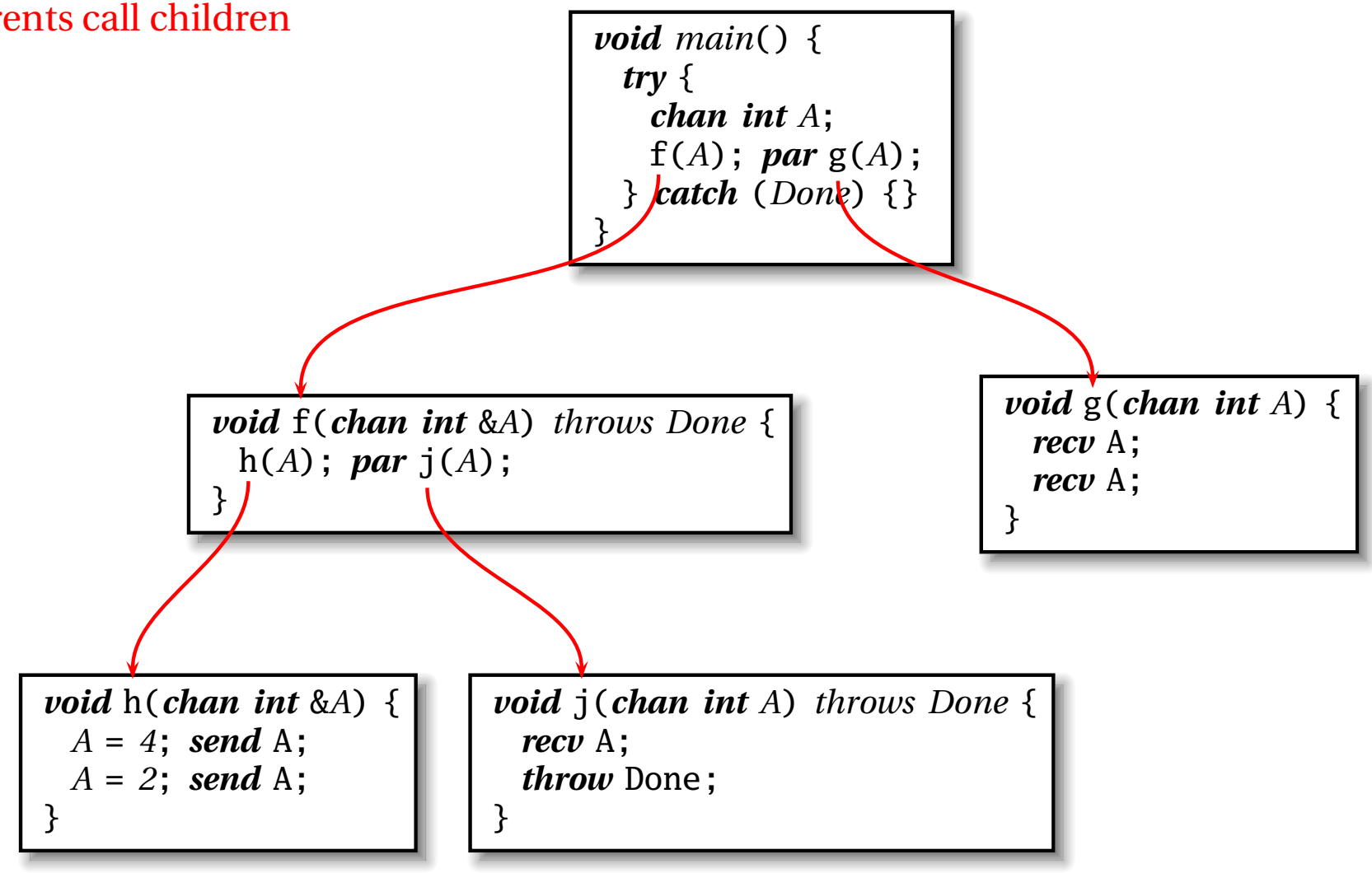
```
void g(chan int A) {
  recv A;
  recv A;
}
```

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

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

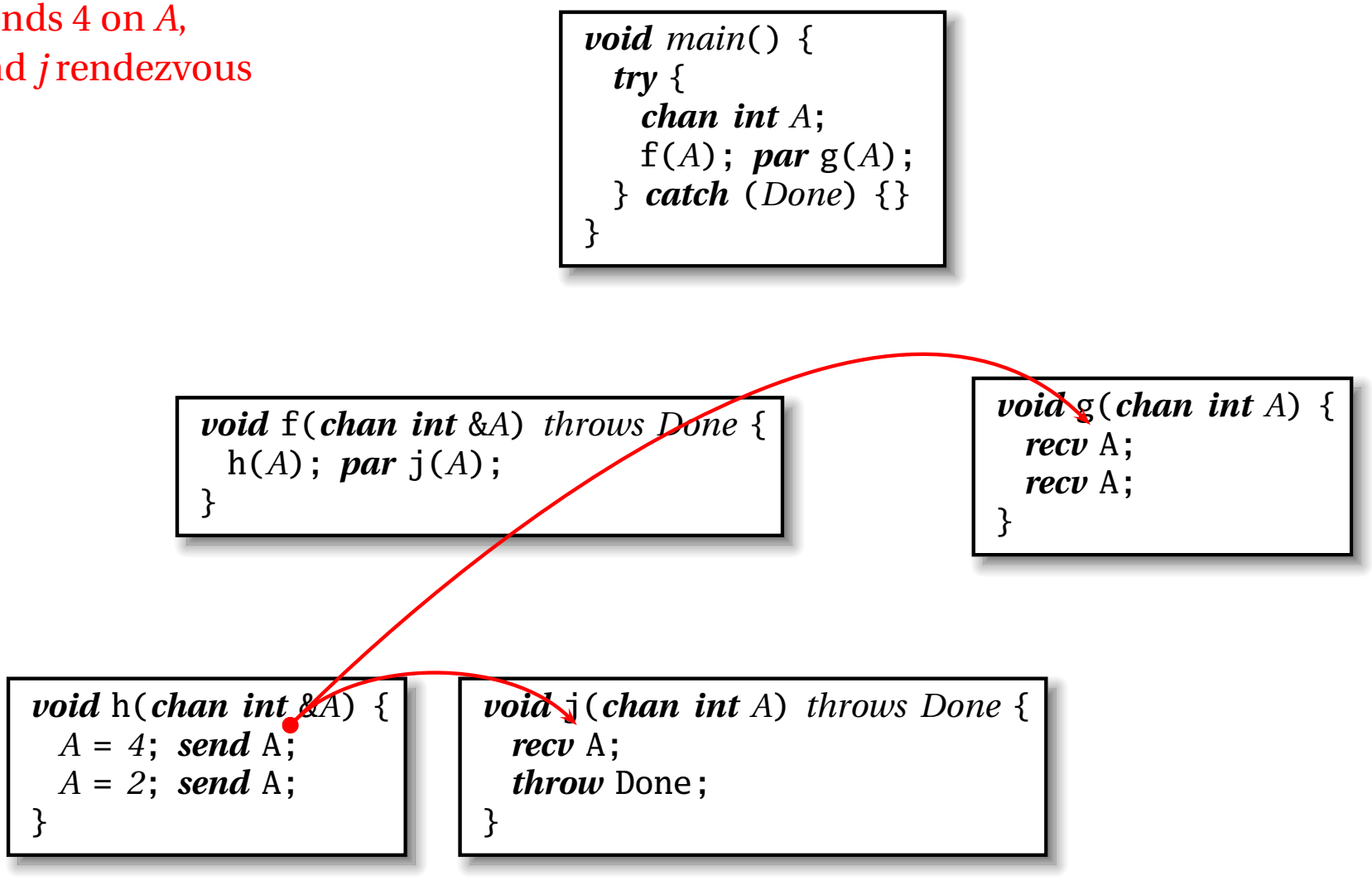
A SHIM example

Parents call children



A SHIM example

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



A SHIM example

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

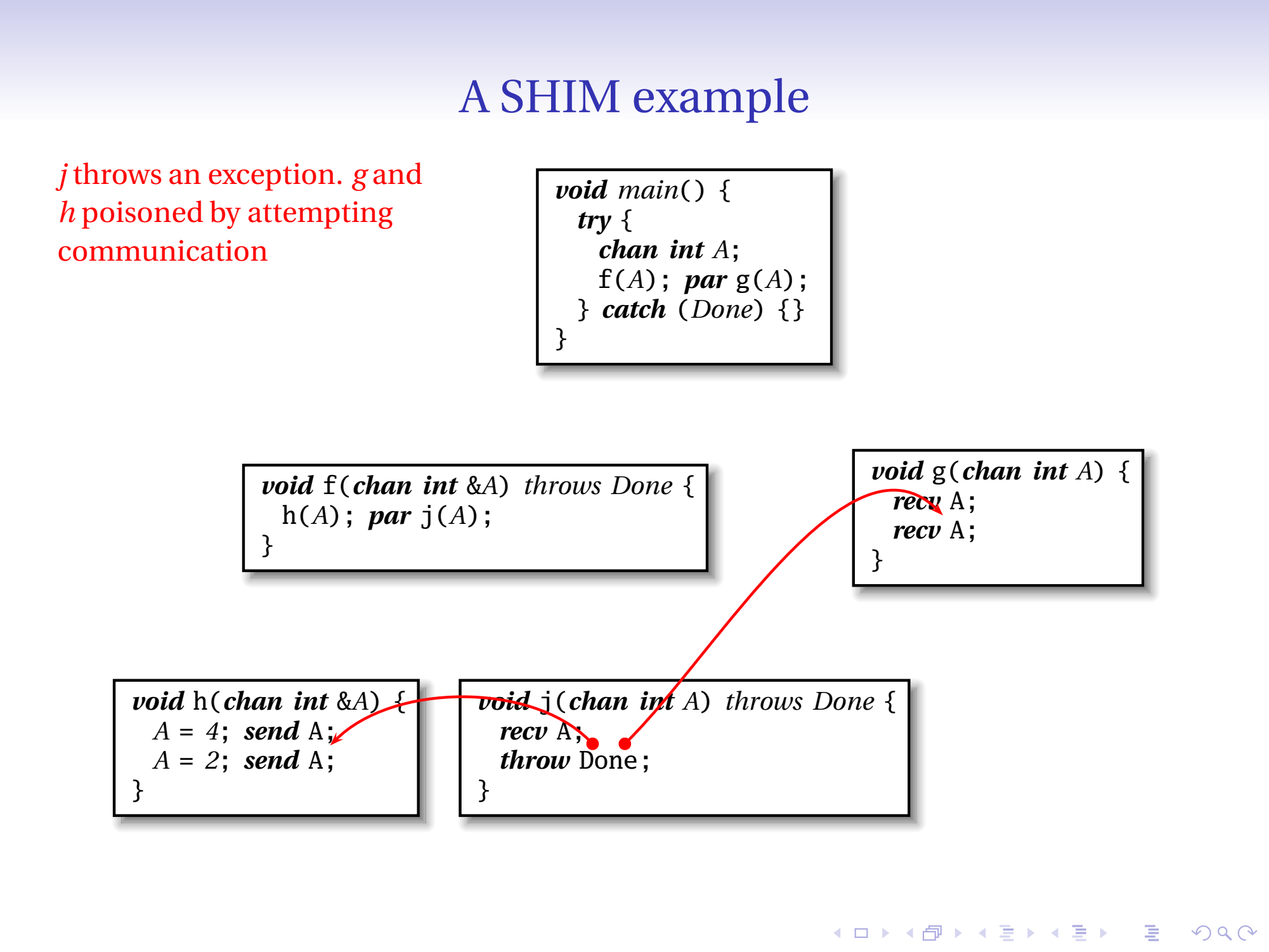
```
void main() {
  try {
    chan int A;
    f(A); par g(A);
  } catch (Done) {}
}
```

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

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

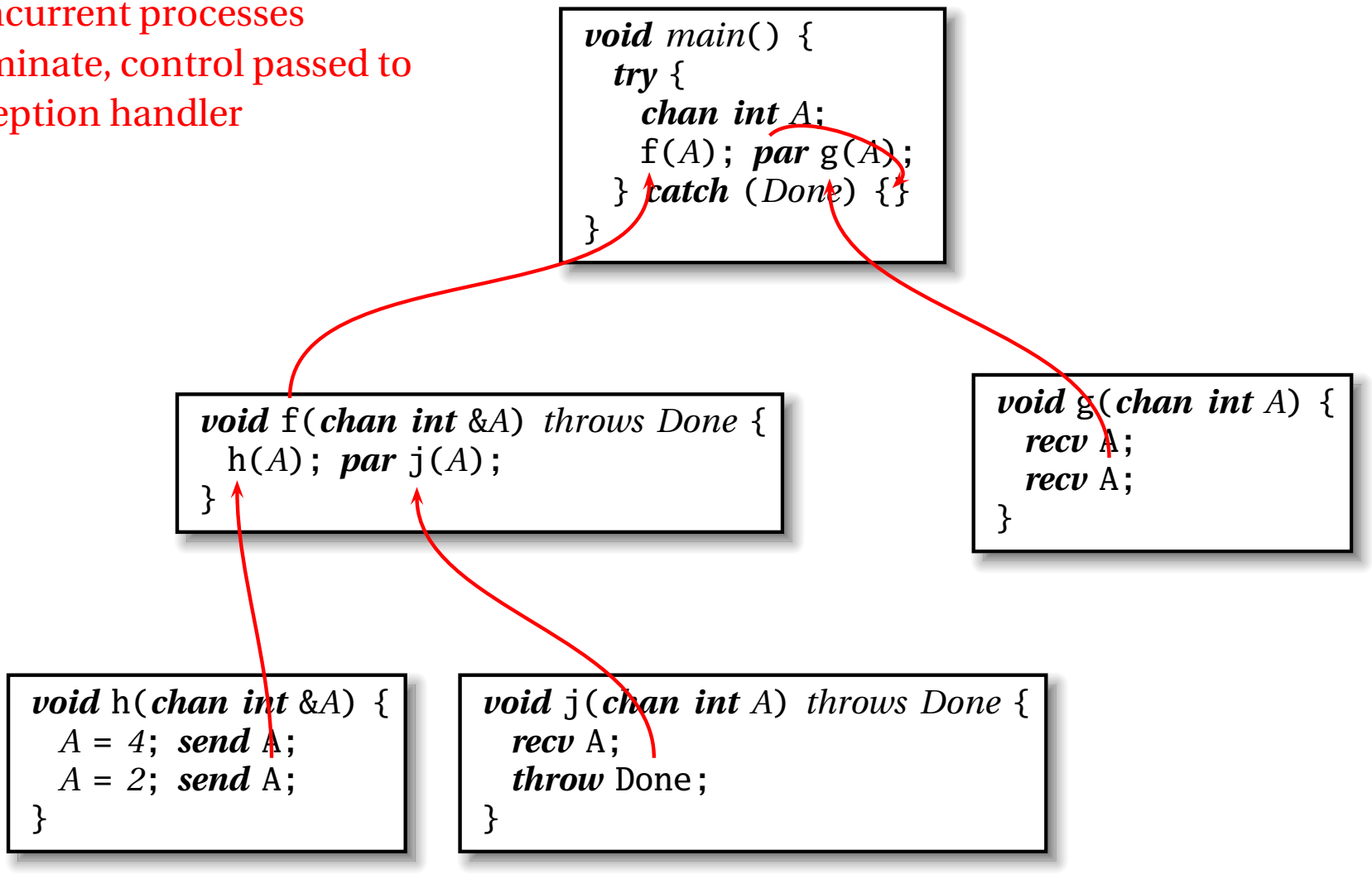
```
void h(chan int &A) {
  A = 4; send A;
  A = 2; send A;
}
```

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



A SHIM example

Concurrent processes terminate, control passed to exception handler



Task and Channel Structures

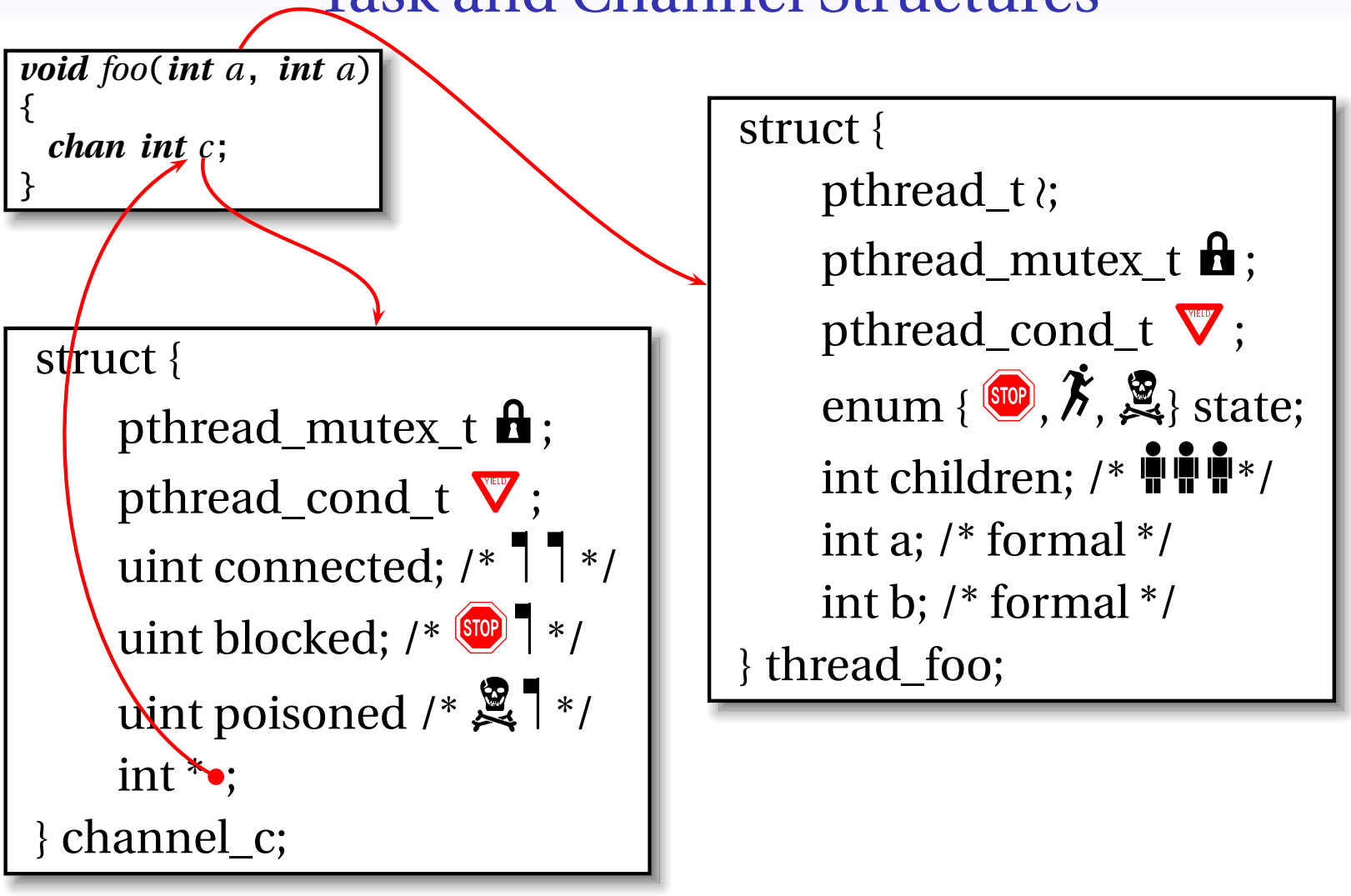
```
void foo(int a, int a)
{
  chan int c;
}
```

Task and Channel Structures

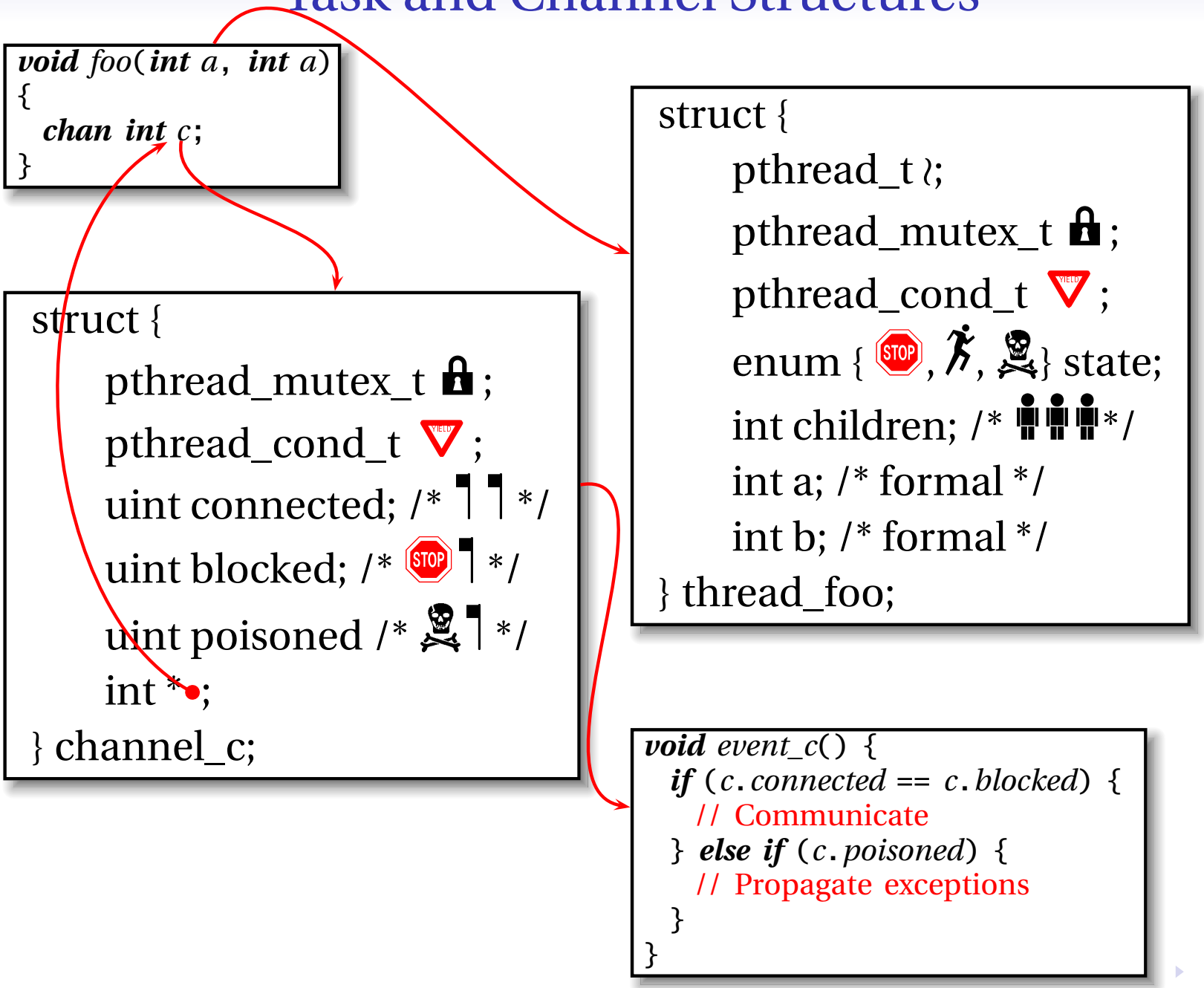
```
void foo(int a, int a)
{
  chan int c;
}
```

```
struct {
        pthread_t ≀;
        pthread_mutex_t 🔒;
        pthread_cond_t YIELD;
        enum { STOP, 🏃, ☠} state;
        int children; /* 👪 */
        int a; /* formal */
        int b; /* formal */
} thread_foo;
```

Task and Channel Structures



Task and Channel Structures



Pthreads Implementation

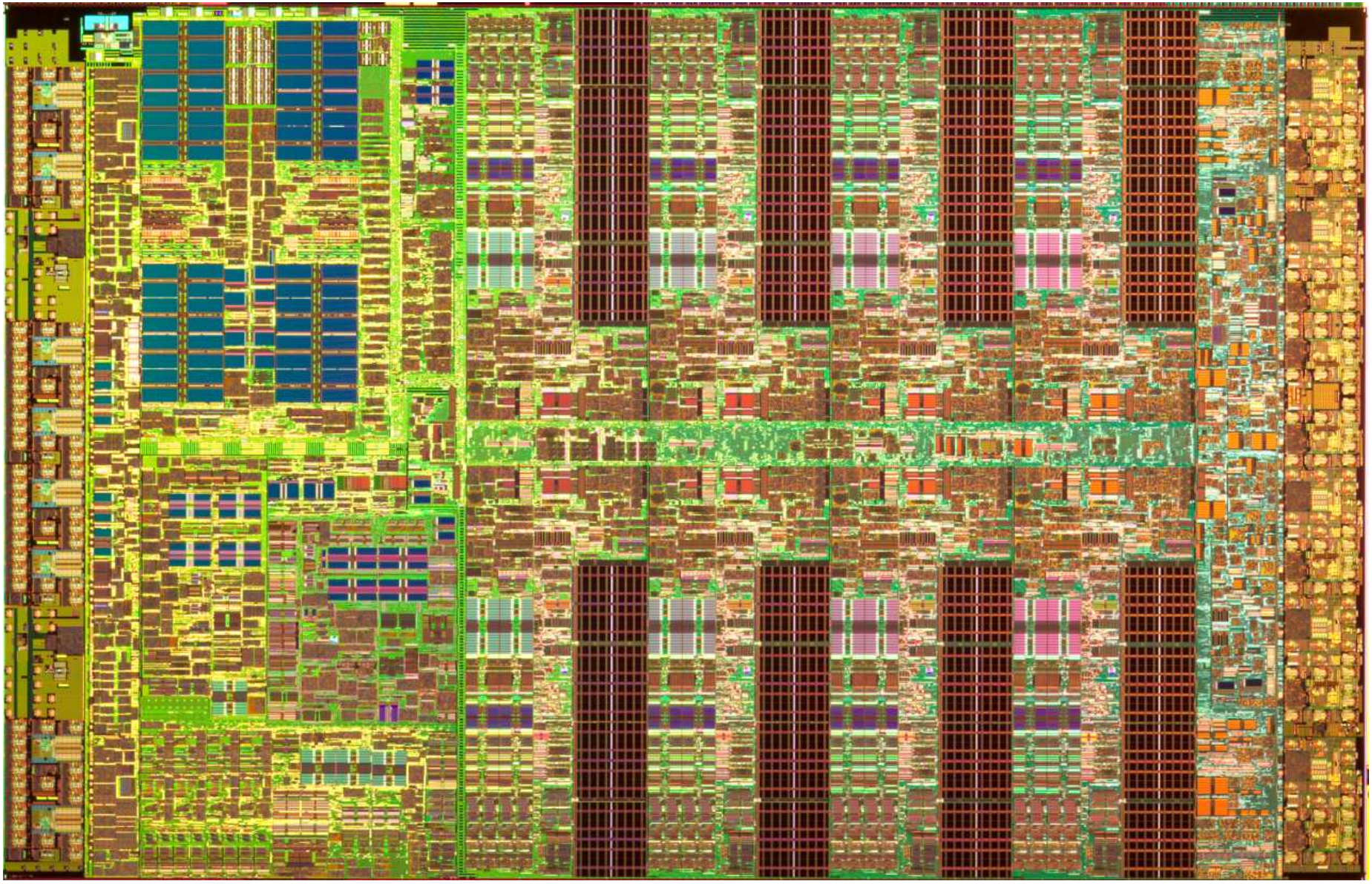
```
void main() {
  try {
    chan int A;
    f(A); par g(A);
  } catch (Done) {}
}

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;
}
```

→

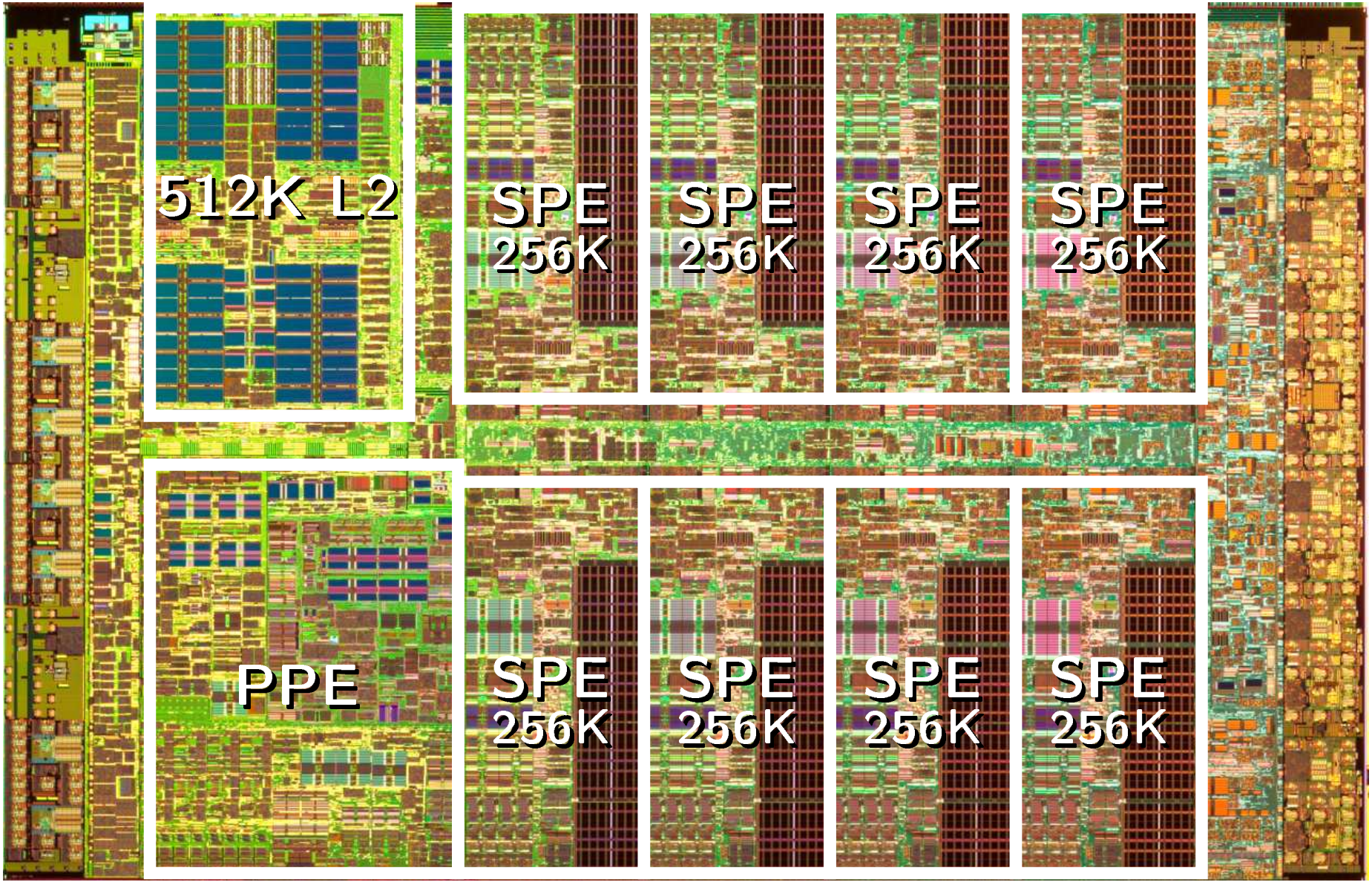
```
struct { ... } _task_main;
void _func_main() { ... } // Code for task main

struct { ... } _chan_A;
void _event_A() { ... } // Synchronize on A


struct { ... } _task_f;
void _func_f() { // Code for task f
}

struct { ... } _task_g;
void _func_g() {
  // Code for task g
}

struct { ... } _task_h;
void _func_h() {
  // Code for task h
}

struct { ... } _task_j;
void _func_j() {
  // Code for task j
}
```


PLAYSTATION 3
SONY

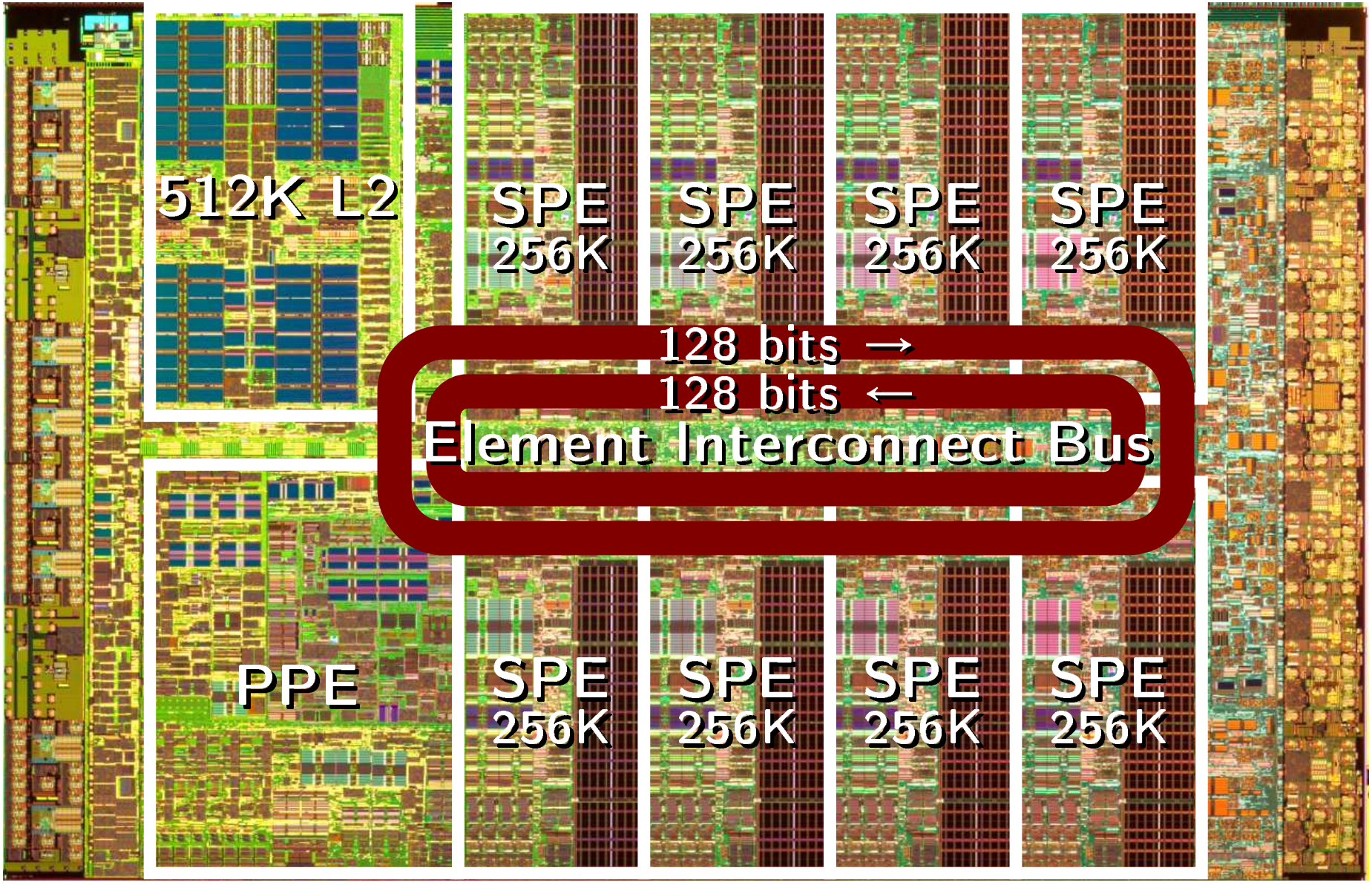
IBM's Cell Broadband Engine



IBM's Cell Broadband Engine



IBM's Cell Broadband Engine



Adapting Pthreads Code to the Cell

```
struct { ... } _task_main;
void _func_main() { ... } // Code for main

struct { ... } _chan_A;
void _event_A() { ... } // Synchronize on A

struct { ... } _task_f;
void _func_f() {
  // Code for task f
}

struct { ... } _task_g;
void _func_g() {
  // Code for task g
}

struct { ... } _task_h;
void _func_h() {
  // Code for task h
}

struct { ... } _task_j;
void _func_j() {
  // Code for task j
}
```

Adapting Pthreads Code to the Cell

PPE Code

```
struct { ... } _task_main;
void _func_main() { ... } // Code for main

struct { ... } _chan_A;
void _event_A() { ... } // Synchronize on A

struct { ... } _task_f;
void _func_f() {
  // Code for task f
}

struct { ... } _task_g;
void _func_g() {
  // Code for task g
}

struct { ... } _task_h;
void _func_h() {
  // Proxy for task h
}

struct { ... } _task_j;
void _func_j() {
  // Proxy for task j
}
```

On SPE 1

```
struct { ... } _task_h;

void main() {
  // Code for task h
}
```

On SPE 2

```
struct { ... } _task_j;

void main() {
  // Code for task j
}
```

Communication Details

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

```
struct {
   ...
   int A;
} _task_j;

void _func_j() { // j's proxy
  mailbox_send(START);
  for (;;) {
    switch (mailbox()) {
      case BLOCK_A:
        _chan_A._blocked |= h;
        _event_A();
        while (_chan_A.blocked & h)
          wait(_chan_A._cond);
        mailbox_send(ACK);
        break;
      case TERM: ...
      case POISON: ...
    }
  }
}
```

```
struct { int A; } _task_j;

void main() { // Code for task j
  for (;;) {
    if (mailbox() == EXIT)
      return;
    DMA_receive(_task_j.A);
    mailbox_send(BLOCK_A);
    if (mailbox() == POISON)
      break;
    DMA_receive(_task_j.A);
    mailbox_send(POISON);
  }
}
```

Communication Details

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

```
struct {
   ...
   int A;
} _task_j;

void _func_j() { // j's proxy
  mailbox_send(START);
  for (;;) {
    switch (mailbox()) {
      case BLOCK_A:
        _chan_A._blocked |= h;
        _event_A();
        while (_chan_A.blocked & h)
          wait(_chan_A._cond);
        mailbox_send(ACK);
        break;
      case TERM: ...
      case POISON: ...
    }
  }
}
```

```
struct { int A; } _task_j;

void main() { // Code for task j
  for (;;) {
    if (mailbox() == EXIT)
      return;
    DMA_receive(_task_j.A);
    mailbox_send(BLOCK_A);
    if (mailbox() == POISON)
      break;
    DMA_receive(_task_j.A);
    mailbox_send(POISON);
  }
}
```

1. Proxy wakes SPE

Communication Details

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

```
struct {
    ...
    int A;
} _task_j;

void _func_j() { // j's proxy
  mailbox_send(START);
  for (;;) {
    switch (mailbox()) {
      case BLOCK_A:
        _chan_A._blocked |= h;
        _event_A();
        while (_chan_A.blocked & h)
          wait(_chan_A._cond);
        mailbox_send(ACK);
        break;
      case TERM: ...
      case POISON: ...
    }
  }
}
```

```
struct { int A; } _task_j;

void main() { // Code for task j
  for (;;) {
    if (mailbox() == EXIT)
      return;
    DMA_receive(_task_j.A);
    mailbox_send(BLOCK_A);
    if (mailbox() == POISON)
      break;
    DMA_receive(_task_j.A);
    mailbox_send(POISON);
  }
}
```

1. Proxy wakes SPE
2. SPE DMAs arguments

Communication Details

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

```
struct {
    ...
    int A;
} _task_j;

void _func_j() { // j's proxy
  mailbox_send(START);
  for (;;) {
    switch (mailbox()) {
      case BLOCK_A:
        _chan_A._blocked |= h;
        _event_A();
        while (_chan_A.blocked & h)
          wait(_chan_A._cond);
        mailbox_send(ACK);
        break;
      case TERM: ...
      case POISON: ...
    }
  }
}
```

```
struct { int A; } _task_j;

void main() { // Code for task j
  for (;;) {
    if (mailbox() == EXIT)
      return;
    DMA_receive(_task_j.A);
    mailbox_send(BLOCK_A);
    if (mailbox() == POISON)
      break;
    DMA_receive(_task_j.A);
    mailbox_send(POISON);
  }
}
```

1. Proxy wakes SPE
2. SPE DMAs arguments
3. SPE blocks on A, notifies proxy

Communication Details

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

```
struct {
    ...
    int A;
} _task_j;

void _func_j() { // j's proxy
  mailbox_send(START);
  for (;;) {
    switch (mailbox()) {
      case BLOCK_A:
        _chan_A._blocked |= h;
        _event_A();
        while (_chan_A.blocked & h)
          wait(_chan_A._cond);
        mailbox_send(ACK);
        break;
      case TERM: ...
      case POISON: ...
    }
  }
}
```

```
struct { int A; } _task_j;

void main() { // Code for task j
  for (;;) {
    if (mailbox() == EXIT)
      return;
    DMA_receive(_task_j.A);
    mailbox_send(BLOCK_A);
    if (mailbox() == POISON)
      break;
    DMA_receive(_task_j.A);
    mailbox_send(POISON);
  }
}
```

1. Proxy wakes SPE
2. SPE DMAs arguments
3. SPE blocks on A, notifies proxy
4. Proxy communicates, notifies SPE

Communication Details

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

```
struct {
    ...
    int A;
} _task_j;

void _func_j() { // j's proxy
  mailbox_send(START);
  for (;;) {
    switch (mailbox()) {
      case BLOCK_A:
        _chan_A._blocked |= h;
        _event_A();
        while (_chan_A.blocked & h)
          wait(_chan_A._cond);
        mailbox_send(ACK);
        break;
      case TERM: ...
      case POISON: ...
    }
  }
}
```

```
struct { int A; } _task_j;

void main() { // Code for task j
  for (;;) {
    if (mailbox() == EXIT)
      return;
    DMA_receive(_task_j.A);
    mailbox_send(BLOCK_A);
    if (mailbox() == POISON)
      break;
    DMA_receive(_task_j.A);
    mailbox_send(POISON);
  }
}
```

1. Proxy wakes SPE
2. SPE DMAs arguments
3. SPE blocks on A, notifies proxy
4. Proxy communicates, notifies SPE
5. SPE DMAs new value

Communication Details

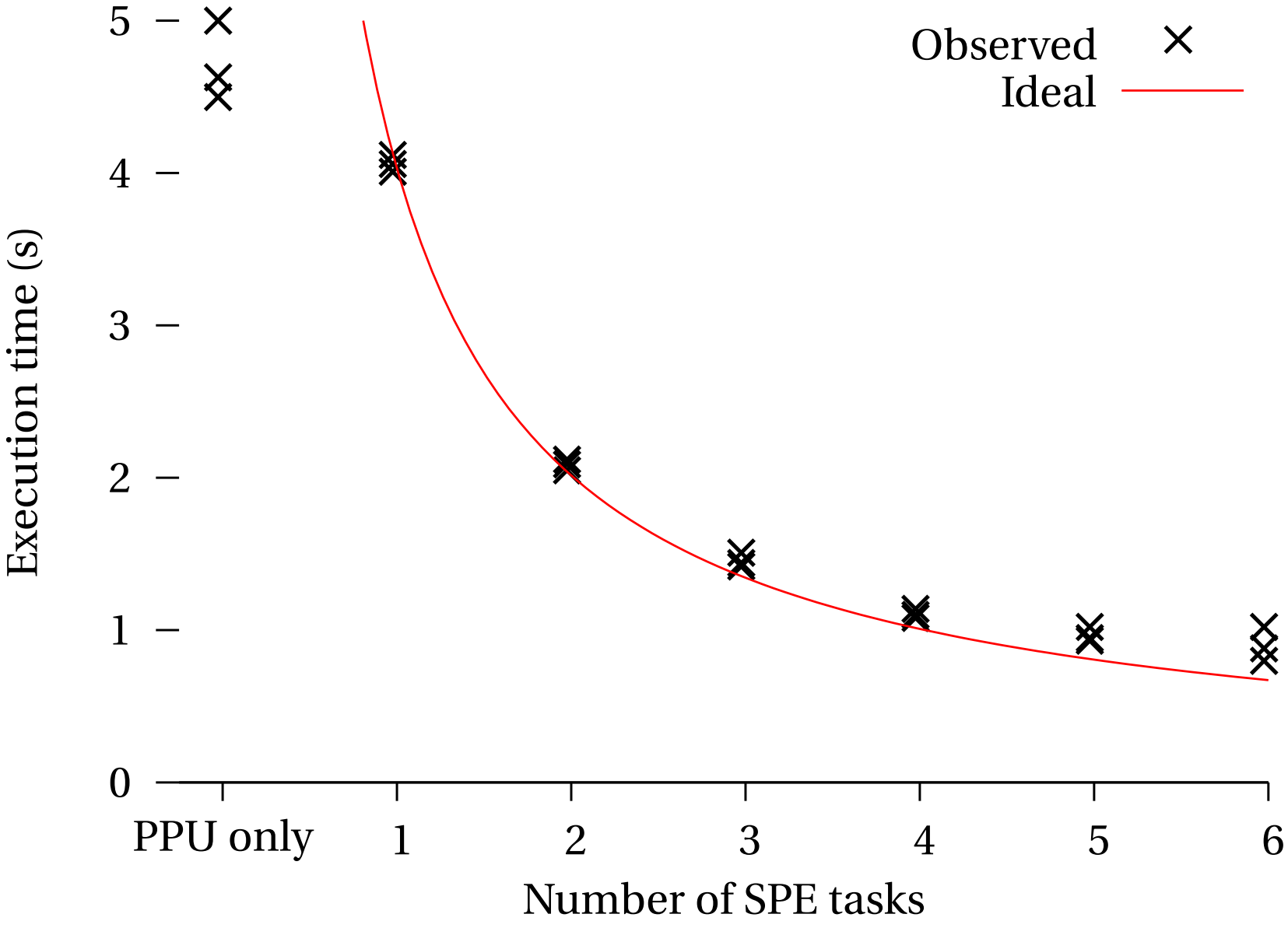
```
void j(chan int A) throws Done {
  recv A;
  throw Done;
}
```

```
struct {
    ...
    int A;
} _task_j;

void _func_j() { // j's proxy
  mailbox_send(START);
  for (;;) {
    switch (mailbox()) {
      case BLOCK_A:
        _chan_A._blocked |= h;
        _event_A();
        while (_chan_A.blocked & h)
          wait(_chan_A._cond);
        mailbox_send(ACK);
        break;
      case TERM: ...
      case POISON: ...
    }
  }
}
```

```
struct { int A; } _task_j;

void main() { // Code for task j
  for (;;) {
    if (mailbox() == EXIT)
      return;
    DMA_receive(_task_j.A);
    mailbox_send(BLOCK_A);
    if (mailbox() == POISON)
      break;
    DMA_receive(_task_j.A);
    mailbox_send(POISON);
  }
}
```

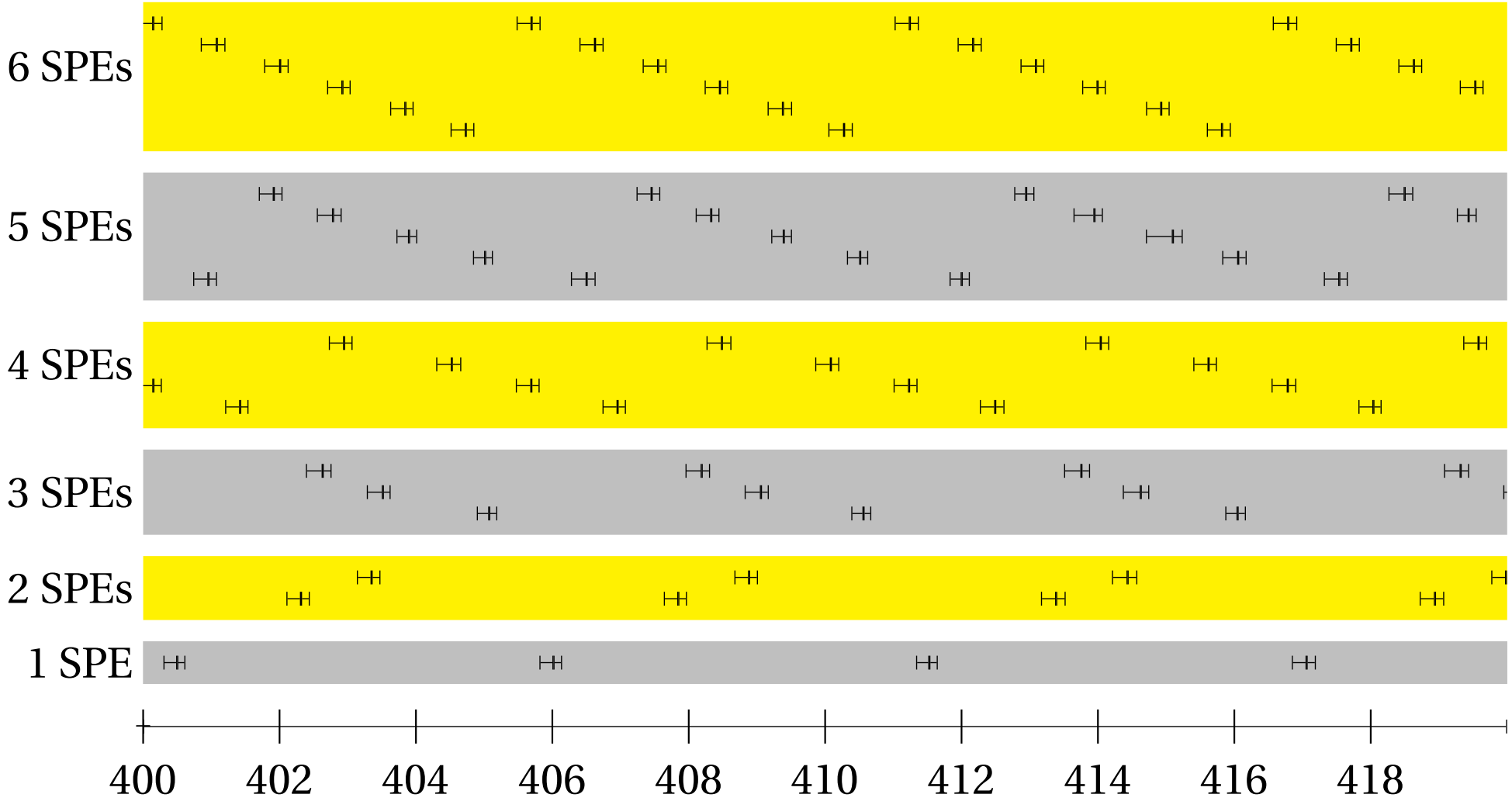
1. Proxy wakes SPE
2. SPE DMAs arguments
3. SPE blocks on A, notifies proxy
4. Proxy communicates, notifies SPE
5. SPE DMAs new value
6. SPE poisons A, notifies proxy

Running Times for the FFT on Varying SPEs

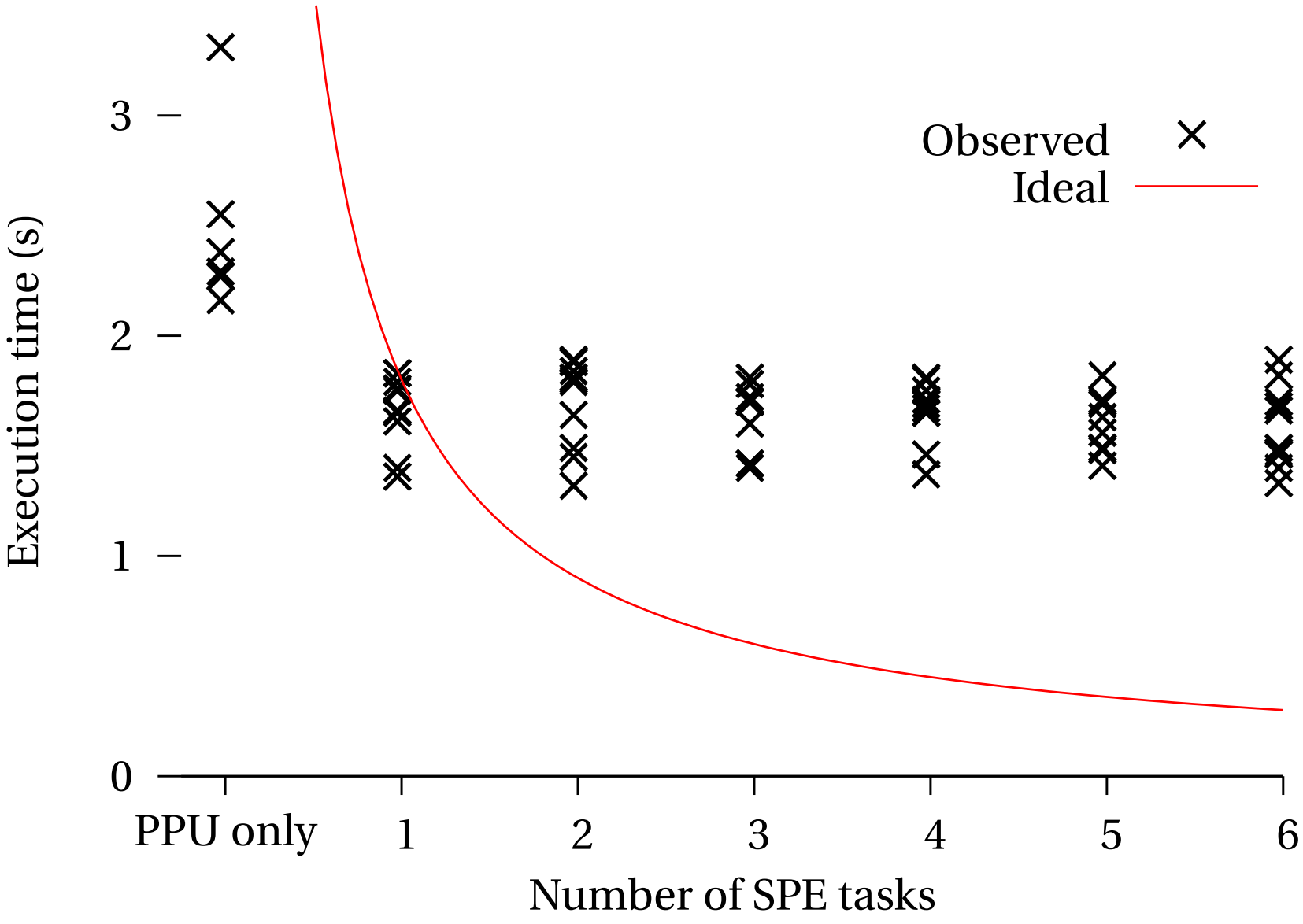


Run on a 20 MB audio file, 1024-point FFTs

Temporal Behavior of the FFT

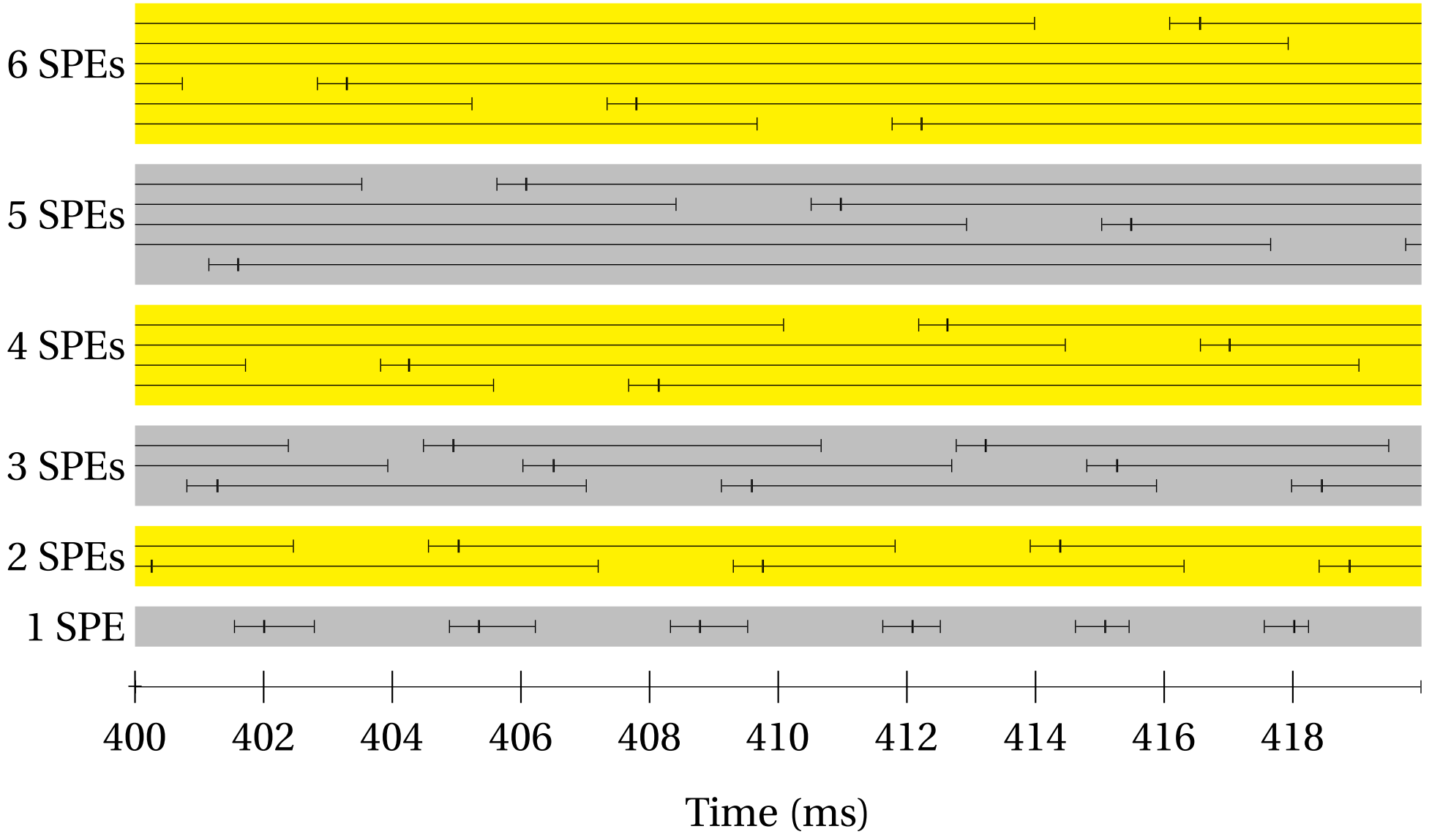


Running Times for the JPEG on Varying SPEs



Run on a 1.7 MB image that expands to a 29 MB raster file

Temporal Behavior of the JPEG Decoder



Conclusions

- SHIM code can be compiled to run on the Cell
  Compiler takes care of synthesizing fussy communication code
- Performance can be excellent for good communication/computation balance
  Near-ideal speedup for embarassingly parallel FFT
- Performance not-so-great when communication outweighs computation
  Amdahl's revenge: sequential part of JPEG dominates
- Need good temporal monitoring tools (not just *gprof*) to get effective speedups.
  SPE performance counters critical; had to be synchronized