Programming Shared Memory Multiprocessors with Deterministic Message-Passing Concurrency: Compiling SHIM to Pthreads

Stephen A. Edwards* Nalini Vasudevan Olivier Tardieu
Department of Computer Science INRIA
Columbia University, New York, USA Sophia Antipolis, France
{sedwards, naliniv}@cs.columbia.edu olivier.tardieu@inria.fr

Abstract

Multicore shared-memory architectures are becoming prevalent and bring many programming challenges. Among the biggest are data races: accesses to shared resources that make a program's behavior depend on scheduling decisions beyond its control. To eliminate such races, the SHIM *concurrent programming language adopts deterministic message passing as it sole communication mechanism.*

We demonstrate such language restrictions are practical by presenting a SHIM *to C-plus-Pthreads compiler that can produce efficient code for shared-memory multiprocessors. We present a parallel* JPEG *decoder and* FFT *exhibiting 3.05 and 3.3× speedups on a four-core processor.*

1 Introduction

Multicore shared-memory multiprocessors now rule the desktop and server markets, and are poised to dominate embedded systems [9]. While such architectures provide high performance per watt, they present many challenges.

Scheduling—instruction ordering—is the biggest issue in programming shared-memory multiprocessors. While uniprocessors go to extremes to provide a sequential execution model despite caches, pipelines, and out-of-order dispatch units, multiprocessors typically only provide such a guarantee for each core in isolation; instructions are at best partially ordered across core boundaries.

Controlling multiprocessor scheduling is crucial not only for performance, but because data races enable scheduling to affect a program's functional behavior. Worse, the operating system schedules nondeterministically.

Many tools attempt to detect races in concurrent programs with shared memory. The SPIN model checker [7] simulates all execution orders of a concurrent program and looks for inconsistencies, Eraser [13] looks for consistent locking behavior, and Atomizer [5] checks atomicity. But these have limitations: SPIN only works on small models; Eraser and Atomizer are testing-based and can miss bugs.

As an alternative, the SHIM language [3, 15] only allows deterministic message-passing communication to guarantee race freedom. The programming model allows SHIM compilers to use a simple syntactic check to verify that runtime scheduling choices cannot change a program's I/O behavior. While this model does restrict how concurrent tasks may interact, the burden for the programmer and the performance penalty are a small price for correctness.

*Edwards and Vasudevan are supported by the NSF, Intel, Altera, the SRC, and NYSTAR.

```
void h(chan int &A) {
  A = 4; send A;
  A = 2; send A;
}
void j(chan int A) throws Done {
  recv A;
  throw Done;
}
void f(chan int &A) throws Done {
  h(A); par j(A);
}
void g(chan int A) {
  recv A;
  recv A;
}
void main() {
  try {
    chan int A;
    f(A); par g(A);
  } catch (Done) {}
}
```

Fig. 1. A concurrent SHIM program with communication and exceptions

Here, we demonstrate SHIM facilitates writing interesting, efficient parallel programs for shared-memory multiprocessors. We implement a parallel JPEG decoder and an FFT to show how SHIM helps with coding and testing different schedules during design exploration (Sec. 3). We present a compiler that generates C code that calls the POSIX thread ("Pthread") library for shared-memory multiprocessors (Sec. 4). For the JPEG and FFT examples, our compiler's output achieves 3.05 and 3.3× speedups on a four-core processor (Sec. 5).

2 The SHIM model and language

SHIM [15] is a concurrent programming language designed to guarantee scheduling independence. The input/output function of a SHIM program does not depend on scheduling choices, i.e., if two concurrent tasks are ready to run, choosing one does not affect the program's function.

It adopts an asynchronous concurrency model, à la Kahn networks [8] (SHIM tasks can only block on a single channel), that uses CSP-like rendezvous [6]. Only communication affects the relative execution rates of concurrent tasks.

The language does not expose shared memory to the programmer, but it does provide single-sender multiple-receiver synchronous communication channels and asynchronous exceptions. Both mechanisms were designed to prevent scheduling decisions from affecting function.

SHIM's syntax is a C subset augmented with constructs for concurrency, communication, and exceptions. It has

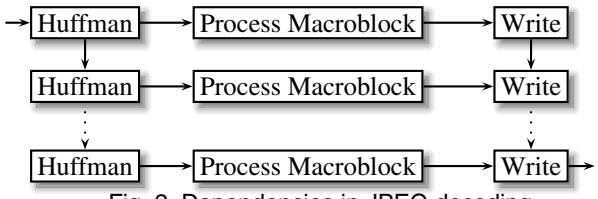

Fig. 2. Dependencies in JPEG decoding

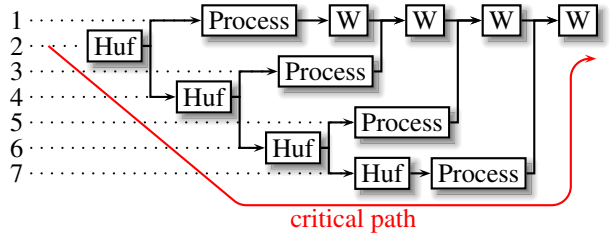

Fig. 3. Seven-task schedule for JPEG

functions with by-value and by-reference arguments, but no global variables, pointers, or recursive types.

The *par* construct starts concurrent tasks. *p par q* starts statements *p* and *q* in parallel, waits for both to complete, then runs the next statement in sequence.

To prevent data races, SHIM forbids a variable to be passed by reference to two concurrent tasks. For example,

```
void f(int &x) {}      void g(int x) {}
void main() {
  int x, y;
  f(x); par g(x); par f(y);                          // OK
  f(x); par f(x);      // rejected because x is passed by reference twice
}
```

Internally, our compiler only supports parallel function calls. If *p* in *p par q* is not a function call, *p* is transformed into a function whose interface—the formal arguments and whether they are by-reference or by-value—is inferred [15].

SHIM's channels enable concurrent tasks to synchronize and communicate without races. The *main* function in Fig. 1 declares the integer channel *A* and passes it to *f* and *g*, then *f* passes it to *h* and *j*. Tasks *f* and *h* send data with *send A*. Tasks *g* and *j* receive it with *recv A*.

A channel resembles a local variable. Passing a channel by value copies its value, which can be modified independently. A channel must be passed by reference to senders.

Communication is blocking: a task that attempts to communicate must wait for all other connected tasks to engage in the communication. If the synchronization completes, the sender's value is broadcast to the receivers. In Fig. 1, 4 is broadcast from *h* to *g* and *j*. Task *g* blocks on the second *send A* because task *j* does not run a matching *recv A*.

Like most formalisms with blocking communication, SHIM programs may deadlock. But deadlocks are easier to fix in SHIM because they are deterministic: on the same input, a SHIM program will either always or never deadlock.

SHIM's exceptions enable a task to gracefully interrupt its concurrently running siblings. A sibling is "poisoned" by

```
void unpack(unpacker_state &state, stripe &stripe) { ... }
void process(const stripe &stripe, pixels &pixels) { ... }
void write(writer_state &wstate, const pixels &pixels) { ... }
unpacker_state ustate; writer_state wstate;
stripe stripe1, stripe2, stripe3, stripe4;
pixels pixels1; chan pixels pixels2, pixels3, pixels4;
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;
} } }
```

Fig. 4. SHIM code for the schedule in Fig. 3

an exception when it attempts to communicate with a task that raised an exception or with a poisoned task. For example, when *j* in Fig. 1 throws *Done*, it interrupts *h*'s blocked *send A* and *g*'s blocked *recv A*. An exception handler runs after all the tasks in its scope have terminated or been poisoned.

3 Design exploration with SHIM

SHIM facilitates the coding and testing of different schedules—a key aspect of design exploration for parallel systems. To illustrate, we describe implementing two parallel algorithms in SHIM: a JPEG decoder and an FFT.

3.1 Porting and parallelizing a JPEG decoder

We started by porting into SHIM a sequential JPEG decoder written in C by Pierre Guerrier. SHIM is not a C subset, so some issues arose. The C code held Huffman tables in global variables, which SHIM does not support, so we passed the tables explicitly. It allocated buffers with *malloc*; we used fixed-size arrays. We discarded a pointer-based Huffman decoder, preferring instead one that used arrays.

After some preprocessing, the main loop of the original program unpacked a macroblock—six Huffman-encoded 8×8 data blocks (standard 4:2:0 downsampling)—performed an IDCT on each data block, converted from YUV to RGB, and blitted the resulting 16×16 pixel block to a framebuffer. It then wrote the framebuffer to a file. Although macroblocks can be processed independently, unpacking and writing are sequential (Fig. 2).

We first ran four IDCT transformers in parallel. Unfortunately, this ran slowly because of synchronization overhead.

To reduce overhead, our next version divided the image into four stripes and processed each independently. Fearing the cost of communication, we devised the seven-task schedule in Fig. 3, which greatly reduced the number of synchronizations at the cost of buffer memory.

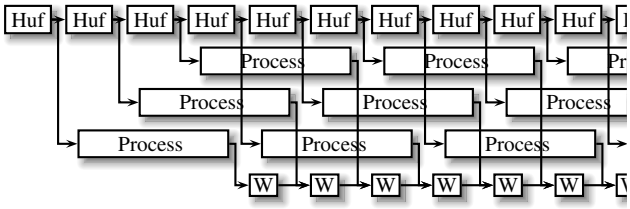

Fig. 5. A pipelined schedule for JPEG

```
void unpack(unpacker_state &state, row &row) { ... }
void process(in row row, out pixels &pixels)
{ for (;;) { recv row; /* IDCT etc. */ send pixels; } }
void write(writer_state wstate, const pixels &pixels) { ... }
unpacker_state ustate; writer_state wstate; int rows;
chan row row1, row2, row3;
chan pixels pixels1, pixels2, pixels3;
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 (;;) {
      recv pixels1; write(wstate, pixels1);
      recv pixels2; write(wstate, pixels2);
      recv pixels3; write(wstate, pixels3); } }
} catch (Done) {}
```

Fig. 6. SHIM code for the JPEG schedule in Fig. 5

The Fig. 3 schedule only gave a 1.8× speedup because the seventh task waits for all the other stripes to be unpacked and then everything waits for the seventh task. The arrow in Fig. 3 shows the critical path, which includes the total cost of Huffman decoding and 1/4 of the IDCTs.

To strike a balance between the two approaches, we finally settled on the more fined-grained schedule in Fig. 5. Each task processes a row of macroblocks at a time (e.g., 64 macroblocks for a 1024-pixel-wide image). This schedule spends less time waiting than the stripe-based approach and synchronizes less often than the block-based approach.

3.2 Parallelizing an FFT

We also coded in SHIM a pipelined FFT to test the effects of numerical roundoff. Its core is the FFT from *Numerical Recipes* [11], which we rewrote to use signed 4.28 fixed-point arithmetic. We added code that parses a .wav file, runs blocks of 1024 16-bit samples through the FFT, through an inverse FFT, then writes the samples to another .wav file.

Our FFT uses a schedule similar to that of the more complex JPEG decoder: one task reads 1024-sample blocks and feeds them to four FFT tasks in a round-robin manner. Each reads its sample block, performs the FFT/inverse FFT operation, and sends its block to a writer task, which receives sample blocks in order and writes them sequentially.

Synchronization costs limited this to a 2.3× speedup on four processors, so we made it process 16 1024-sample blocks, improving performance to 3.3×.

3.3 Race freedom

Both the JPEG and FFT examples illustrate that dividing and scheduling computation tasks is critical in achieving performance on parallel hardware. Although data dependencies in JPEG were straightforward, finding the right schedule took some effort. With traditional concurrent formalisms, it is easy to introduce data races during design exploration.

SHIM's channels and exceptions cannot introduce races. E.g., in Fig. 6, the first task throws an exception after reading all the rows. SHIM semantics ensure that the three row-processing tasks and the writing task terminate just after they have completed processing all the rows.

SHIM also guarantees data dependencies are respected. For instance, the SHIM compiler reject attempts to run unpackers in parallel because of the shared pass-by-reference state (mostly, position in the file):

```
void unpack(unpacker_state &state, stripe &stripe) { ... }
unpack(ustate, stripe1); par unpack(ustate, stripe2);      // rejected
```

4 Generating Pthreads code for SHIM

In this section, we describe our main technical contribution: a SHIM compiler that generates parallel C code that uses the Pthread library's threads (independent program counters and stacks that share program and data memory), mutexes (mutual exclusion objects for synchronizing access to shared memory), and condition variables (can block and resume execution of other threads).

4.1 Mutexes and condition variables

Any Pthreads program must decide how many threads it will use, the number of mutexes, the partition of shared state, and the number and meaning of condition variables. These are partly engineering questions: coarse-grain locking leads to fewer locking operations but more potential for contention; finer locking has more overhead. Locking is fairly cheap, typically consisting of a (user-space) function call containing an atomic test-and-set instruction, but is not free. On one machine, locking and unlocking a mutex took 74× as long as a floating point multiply-accumulate.

We generate code that uses one mutex/condition variable pair for each task and for each channel. Fig. 7 shows the data structures we use. These are "base classes:" the type of each task and channel includes additional fields that hold the formal arguments passed to the task and, for each function to which a channel is passed by value, a pointer to the local copy of the channel's value. To reduce locking, we track exception "poisoning" in both tasks and channels.

4.2 The static approach

For efficiency, our compiler assumes the communication and call graph of the program is static. We reject programs with recursive calls, allowing us to transform the call graph into a call tree. This duplicates code to improve performance: fewer channel aspects are managed at run time.

```
#define lock(m) pthread_mutex_lock(&m)
#define unlock(m) pthread_mutex_unlock(&m)
#define wait(c, m) pthread_cond_wait(&c, &m)
#define broadcast(c) pthread_cond_broadcast(&c)
enum state { STOP, RUN, POISON };
struct task {
  pthread_t thread;
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  enum state state;
  unsigned int attached_children;
  /* Formal arguments... */
};
struct channel {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  unsigned int connected;
  unsigned int blocked;
  unsigned int poisoned;
  /* Local copy pointers... */
};
```

Fig. 7. Shared data structures for tasks and channels

```
lock(A.mutex);                          /* acquire lock for channel A */
A.blocked |= (A_h|A_f|A_main);          /* block h and ancestors on A */
event_A();                              /* alert channel of the change */
while (A.blocked & A_h) {               /* while h remains blocked */
  if (A.poisoned & A_h) {               /* were we poisoned? */
    unlock(A.mutex); goto _poisoned;
  }
  wait(A.cond, A.mutex);                /* wait on channel A */
}
unlock(A.mutex);                        /* release lock for channel A */
```

Fig. 8. C code for *send A* in function *h()*

We encode in a bit vector the subtree of functions connected to a channel. Since we know at compile time which functions can connect to each channel, we assign a unique bit to each function on a channel. We check these bits at run time with logical mask operations. In the code, something like *A_f* is a constant that holds the bit our compiler assigns to function *f* connected to channel *A*, such as 0x4.

4.3 Implementing rendezvous communication

Implementing SHIM's multiway rendezvous communication with exceptions is the main code generation challenge.

The code at a send or receive is straightforward: it locks the channel, marks the function and its ancestors as blocked, calls the *event* function for the channel to attempt the communication, and blocks until communication has occurred. If it was poisoned, it branches to a handler. Fig. 8 is the code for *send A* in *h* in Fig. 1.

For each channel, our compiler generates an *event* function that manages communication. Our code calls an *event* function when the state of a channel changes, such as when a task blocks or connects to a channel.

Fig. 9 shows the *event* function our compiler generates for channel *A* in Fig. 1. While complex, the common case is quick: when the channel is not ready (one connected task is not blocked on the channel) and no task is poisoned, *A.connected* != *A.blocked* and *A.poisoned* == *0* so the bodies of the two *if* statements are skipped.

If the channel is ready to communicate, *A.blocked* == *A.connected* so the body of the first *if* runs. This clears the channel (*blocked* = *0*) and *main*'s value for *A* (passed by reference to *f* and *h*) is copied to *g* or *j* if connected.

If at least one task connected to the channel has been poisoned, *A.poisoned* != *0* so the body of the second *if* runs.

```
void event_A() {
  unsigned int can_die = 0, kill = 0;
  if (A.connected == A.blocked) {                    /* communicate */
    A.blocked = 0;
    if (A.connected & A_g) *A.g = *A.main;
    if (A.connected & A_j) *A.j = *A.main;
    broadcast(A.cond);
  } else if (A.poisoned) {                           /* propagate exceptions */
    can_die = blocked & (A_g|A_h|A_j);               /* compute can_die set */
    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 set */
      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) {             /* poison some tasks? */
      unlock(A.mutex);
      if (kill & A_g) {                              /* poison g if in kill set */
        lock(g.mutex);
          g.state = POISON;
        unlock(g.mutex); }
      /* also poison f, h, and j if in kill set... */
      lock(A.mutex);
        A.poisoned |= kill; broadcast(A.cond);
} } }
```

Fig. 9. C code for the *event* function for channel *A*

```
lock(main.mutex); main.state = POISON; unlock(main.mutex);
lock(f.mutex); f.state = POISON; unlock(f.mutex);
lock(j.mutex); j.state = POISON; unlock(j.mutex);
goto _poisoned;
```

Fig. 10. C code for *throw Done* in function *j()*

This code comes from unrolling a recursive procedure at compile time, which is possible because we know the structure of the channel (i.e., which tasks connect to it). The speed of such code is a key advantage over a library.

This exception-propagation code attempts to determine which tasks, if any, connected to the channel should be poisoned. It does this by manipulating two bit vectors. A task *can_die* iff it is blocked on the channel and all its children connected to the channel (if any) also *can_die*. A *poisoned* task may *kill* its sibling tasks and their descendants. Finally, the code kills each task in the *kill* set that *can_die* and was not *poisoned* before by setting its *state* to *POISON* and updating the channel accordingly (*A.poisoned* |= *kill*).

Code for throwing an exception (Fig. 10) marks as *POISON* all its ancestors up to where it will be handled. Because the compiler knows the call tree, it knows how far to "unroll the stack," i.e., how many ancestors to poison.

4.4 Starting and terminating tasks

It is costly to create and destroy a POSIX thread because it usually requires a system call, each has a separate stack, and doing so interacts with the operating system's scheduler. To minimize this overhead, because we know the call graph of the program at compile time, our compiler generates code that creates at the beginning as many threads as the SHIM program will ever need. These threads are only destroyed when the SHIM program terminates; if a SHIM task terminates, its POSIX thread blocks until it is re-awakened.

```
lock(A.mutex); /* connect */
  A.connected |= (A_f|A_g);
  event_A();
unlock(A.mutex);
lock(main.mutex);
  main.attached_children = 2;
unlock(main.mutex);
lock(f.mutex); /* pass args */
  f.A = &A;
unlock(f.mutex);

/* A is dead on entry for g,
   so do not pass A to g */
lock(f.mutex);          /* run f() */
  f.state = RUN; broadcast(f.cond);
unlock(f.mutex);
lock(g.mutex);          /* run g() */
  g.state = RUN; broadcast(g.cond);
unlock(g.mutex);
lock(main.mutex); /* wait for children */
  while (main.attached_children)
    wait(main.cond, main.mutex);
  if (main.state == POISON) {
    unlock(main.mutex);
    goto _poisoned; }
unlock(main.mutex);
```

Fig. 11. C code for calling *f()* and *g()* in *main()*

```
int *A; /* value of channel A */

_restart:
 lock(f.mutex);
   while (f.state != RUN)
     wait(f.cond, f.mutex);
   A = f.A; /* copy arg. */
 unlock(f.mutex);

 /* body of the f task */

_terminated:
 lock(A.mutex); /* disconnect f */
   A.connected &= ~A_f;
   event_A();
 unlock(A.mutex);
 lock(f.mutex); /* stop */
   f.state = STOP;
 unlock(f.mutex);
 goto _detach;

_poisoned:
 lock(A.mutex); /* poison A */
   A.poisoned |= A_f;
   A.blocked &= ~A_f;
 event_A();
 unlock(A.mutex);
 lock(f.mutex); /* wait for children */
   while (f.attached_children)
     wait(f.cond, f.mutex);
 unlock(f.mutex);
 lock(A.mutex); /* disconnect j, h */
   A.connected &= ~(A_h|A_j);
   A.poisoned &= ~(A_h|A_j);
   event_A();
 unlock(A.mutex);
_detach: /* detach from parent */
 lock(main.mutex);
   --main.attached_children;
   broadcast(main.cond);
 unlock(main.mutex);
 goto _restart;
```

Fig. 12. C code in function *f()* controlling its execution

Fig. 11 shows the code in *main* that runs f and g in parallel. It connects f and g to channel A, sets its number of live children to 2, passes function parameters, then starts f and g. The address for the pass-by-reference argument A is passed to f. Normally, a value for A would be passed to g, but our compiler found this value is not used so the copy is avoided (discussed below). After starting f and g, *main* waits for both children to return. Then *main* checks whether it was poisoned, and if so, branches to a handler.

Reciprocally, Fig. 12 shows the code in f that controls its execution: an infinite loop that waits for *main*, its parent, to set its *state* field to running, at which point it copies its formal arguments into local variables and runs its body.

If a task terminates normally, it cleans up after itself. In Fig. 12, task f disconnects from channel A, sets its *state* to *STOP*, and informs *main* it has one less running child.

By contrast, if a task is poisoned, it may still have children running and it may also have to poison sibling tasks so it cannot entirely disappear yet. In Fig. 12, task f, if poisoned, does not disconnect from A but updates its *poisoned* field. Then, task f waits for its children to return. At this time, f can disconnect its (potentially poisoned) children from channels, since they can no longer poison siblings. Finally, f informs *main* it has one less running child.

4.5 Optimizations

SHIM draws no distinction between sequential C-like functions and concurrent tasks; our compiler treats them differently for efficiency. Our compiler distinguishes tasks from functions, which must not take any channel arguments, contain local channels, throw or handle exceptions, have parallel calls, call any tasks, or be called in parallel. Tasks are implemented as described above—each is assigned its own thread. Functions follow C's calling conventions.

Unlike Java, SHIM passes scalars, structures, and arrays by value unless marked as by-reference. This is convenient at parallel call sites to avoid interference among concurrent tasks. However, if tasks only read some data, the data can be shared among them for efficiency. Similarly, a channel can be shared among tasks that never update the channel's value between *recv* instructions. We introduced a C++-like *const* specifier that prohibits assignments to a variable, channel, or function parameter. The compiler allows multiple concurrent *const* by-reference parameters and allocates a shared copy for *const* parameters passed by value.

We implemented another optimization to reduce superfluous copies of large data structures. Normally, the current value of a channel is copied when the channel is passed by value, but copying is unnecessary if the value is never used before the next value is *recv*'d. The overhead can be substantial for arrays. We perform live variable analysis to determine which arguments are dead on entry. E.g., in

```
void myfunc(chan int input[65536]) { recv input; ... }
```

the *input* channel value is dead on entry and will not be copied at any callsite for *myfunc*, eliminating a 256K copy.

5 Experimental results

We implemented our SHIM compiler in OCAML. Code specific to the Pthreads backend is only about 2000 lines.

To test the performance of our generated code, we ran it on a 1.6 GHz Quad-Core Intel Xeon (E5310) server running Linux kernel 2.6.20 with SMP (Fedora Core 6). The processor "chip" actually consists of two dice, each containing a pair of processor cores. Each core has a 32 KB L1 instruction and a 32 KB L1 data cache, and each die has a 4 MB of shared L2 cache shared between the two cores.

We compiled the generated C with gcc 4.1.1 with *-O7* and *-pthread* options. We timed it using the *time* command and ran *sync* to flush the disk cache. We used Dag Wieers's *run* command[1] to limit the number of available cores. The JPEG program uses much more stack space than typical C programs because it stores all data on the stack instead of the heap. We raised the stack size to 16 MB with *ulimit -s*.

Table 1 shows results for the JPEG decoder. We ran it on a 20 MB earth image from NASA[2] and varied both the number of available processors and the number of row-processing

[1] www.ccur.com/oss

[2] world.200409.3x21600x10800.jpg from earthobservatory.nasa.gov

Table 1. Experimental Results for the JPEG decoder

Cores	Tasks	Time	Total	Total/Time	Speedup
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

† Reference single-threaded C implementation.
Run on a 20 MB 21600 × 10800 image that expands to 668 MB. Tasks is the number of parallel threads (read and unpack + process row + write), Time is wallclock, Total is user + system time, Total/Time is the parallelization factor, speedup is with respect to the reference implementation.

tasks in our program. The speedup due to parallelization plateaued at 3.05, which we attribute to the sequential nature of the Huffman decoding process.

Table 2 shows statistics for our FFT. We compared handwritten C with sequential SHIM and two parallel SHIM versions, one with six tasks that work on single 1024-sample blocks and one that works on sixteen such blocks. The first parallel implementation has overhead from synchronization and communication. The "Parallel 16" version communicates less to reduce this overhead and achieve a 3.3× speedup: 82% of an ideal 4× speedup on four cores.

6 Related work

Like SHIM, the StreamIt language [16] is deterministic, but its dataflow model is a strict subset of SHIM's and there is no StreamIt compiler for shared memory machines.

Other concurrent languages use different models. The most common is "loops-over-arrays," embodied, e.g., in compilers for OpenMP [17]. This would be awkward for a schedule such as Fig. 5. The Cilk language [2] speculates to parallelize sequential code. The Guava [1] Java dialect prevents unsynchronized access to shared objects by enforcing monitor use with a type system. Like SHIM, it aims for race freedom, but uses a very different model.

7 Conclusions

A good parallel algorithm reliably computes the result quickly. Unlike most parallel languages, SHIM guarantees reliability by preventing data races. Correctness remains a challenge, but at least running a SHIM program on a test case gives consistent results for any scheduling policy.

SHIM is helpful during design exploration when testing different schedules; its determinacy makes it easy to obey data dependencies. Its C-like syntax facilitates porting existing code. We demonstrated this on a JPEG decoder.

Our SHIM compiler generated code for parallel programs that runs on a four-core processor over three times faster than sequential C. Sequential SHIM code runs no slower.

Table 2. Experimental Results for the FFT

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

Run on a 40 MB audio file—20 000 1024-point FFTs.

Our plans for SHIM include formal verification, code generation fusing parallelism with static scheduling [4], extracting parallelism [10], data distribution [14], communication optimization [12], and the synthesis of hardware.

[1] D. F. Bacon et al. Guava: A dialect of Java without data races. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 382–400, Minneapolis, Minnesota, Oct. 2000.

[2] R. D. Blumofe et al. Cilk: An efficient multithreaded runtime system. In *Principles and Practice of Parallel Programming (PPoPP)*, pp. 207–216, Santa Barbara, CA, July 1995.

[3] S. A. Edwards and O. Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. In *Embedded Software (Emsoft)*, pp. 37–44, Jersey City, NJ, Sept. 2005.

[4] S. A. Edwards and O. Tardieu. Efficient code generation from SHIM models. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pp. 125–134, Ottawa, Canada, June 2006.

[5] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. Principles of Programming Languages (POPL)*, pp. 256–267, Venice, Italy, Jan. 2004.

[6] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

[7] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

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

[9] D. McGrath. Intel rolls quad-core CPUs for embedded computing. *EE Times*, April 3 2007.

[10] S. Meijer, B. Kienhuis, A. Turjan, and E. de Kock. A process splitting transformation for Kahn process networks. In *Design, Automation, and Test in Europe (DATE)*, pp. 1355–1360, Nice, France, Apr. 2007.

[11] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, 1988.

[12] J. Reppy and Y. Xiao. Specialization of CML message-passing primitives. *SIGPLAN Notices*, 42(1):315–326, 2007.

[13] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

[14] P. Stanley-Marbell et al. Adaptive data placement in an embedded multiprocessor thread library. In *Design, Automation, and Test in Europe (DATE)*, pp. 698–699, Munich, Germany, Mar. 2006.

[15] O. Tardieu and S. A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *Embedded Software (Emsoft)*, pp. 142–151, Seoul, Korea, Oct. 2006.

[16] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Compiler Construction (CC)*, volume 2304 of *LNCS*, pp. 179–196, Grenoble, France, Apr. 2002.

[17] X. Tian, M. Girkar, A. Bik, and H. Saito. Practical compiler techniques on efficient multithreaded code generation for OpenMP programs. *The Computer Journal*, 48(5):588–601, 2005.