Celling SHIM: Compiling Deterministic Concurrency to a Heterogeneous Multicore

Nalini Vasudevan
Columbia University
New York, New York
naliniv@cs.columbia.edu

Stephen A. Edwards
Columbia University
New York, New York
sedwards@cs.columbia.edu

ABSTRACT

Parallel architectures are the way of the future, but are notoriously difficult to program. In addition to the low-level constructs they often present (e.g., locks, DMA, and non-sequential memory models), most parallel programming environments admit data races: the environment may make nondeterministic scheduling choices that can change the function of the program.

We believe the solution is model-based design, where the programmer is presented with a constrained higher-level language that prevents certain unwanted behavior. In this paper, we describe a compiler for the SHIM scheduling-independent concurrent language that generates code for the Cell Broadband heterogeneous multicore processor. The complexity of the code our compiler generates relative to the source illustrates how difficult it is to manually write code for the Cell.

We demonstrate the efficacy of our compiler on two examples. While the SHIM language is (by design) not ideal for every algorithm, it works well for certain applications and simplifies the parallel programming process, especially on the Cell architecture.

Categories and Subject Descriptors

D.3.4 [**Software**]: Programming Languages—*Processors*

General Terms

Languages, Performance

Keywords

Cell Processor, SHIM, Parallelism, Concurrency, Compiler

1. INTRODUCTION

Traditional processor architecture has hit a power/performance wall; we cannot expect any improvements in single-threaded performance [2]. Task-level parallelism appears to be the way forward for mainstream and embedded systems alike.

Parallel software, unfortunately, presents a much larger design challenge than traditional sequential software. Parallelism typically adds data races, deadlocks, non-atomic data structure updates, priority inversion, nondeterminism, and a host of other problems to the already-substantial challenges of designing complex software.

We believe the way forward is to provide more constrained parallel programming models that avoid some of these challenges by construction. While restrictions mean that certain kinds of algorithms are ruled out (or at least very awkward to code), this seems a reasonable price to pay for correct programs.

We present a compiler for the SHIM scheduling-independent concurrent programming language [6] that produces code for the Cell heterogeneous multicore processor. In contrast to the existing programming environment, which requires two C compilers, explicitly coded DMA transfers, numerous alignment directives, different C code for each kind of core, and can produce programs that behave nondeterministically, our SHIM environment for the Cell guarantees determinism, provides a simple, robust communication mechanism, and allows different partitions with no source code changes.

Below, we review the SHIM language, the Cell processor, and describe the inner workings of our compiler. In Section 5, we describe how we instrumented our generated code to collect performance data, and present experimental results in Section 6.

2. THE SHIM LANGUAGE

The SHIM language [6,25] promotes model-based design by providing a restricted model of computation that renders it difficult or impossible to make certain kinds of design errors. It is intended for concurrent software, which is harder to reason about than sequential because of data races, non-atomic accesses, deadlocks, memory consistency issues, and nondeterminism.

SHIM guarantees scheduling independence: any nondeterministic scheduling choices made at run time cannot affect the function of a program. It eliminates data races, and it also simplifies formal analysis. For example, it is easy to translate a SHIM program into a synchronous automaton for model-checking [27].

SHIM gains scheduling independence by adopting stream-based dataflow semantics inspired by Kahn networks [16]. However, it eschews Kahn's unbounded buffers as impractical, adopting instead CSP-inspired multiway rendezvous [12]. Synchronous dataflow [18] and StreamIt [26] also adopt Kahn semantics, but SHIM is more expressive: it supports data-dependent communication patterns.

SHIM uses a C-like syntax augmented with constructs for concurrency, communication, and exceptions [25]. It has functions with by-value and by-reference arguments. SHIM prohibits global variables and pointers and currently does not support recursive types.

The p ***par*** q construct 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 passing a variable by reference to two concurrent tasks. For example,

SAC '09 March 8–12, 2009, Honolulu, Hawaii, USA

```
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
}
```

If, in p ***par*** q, p is not a function call, our compiler transforms it into a function whose interface is inferred [25].

SHIM's channels enable tasks to synchronize and communicate without races. The *main* function in Figure 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 a sender.

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 Figure 1, the value 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 at least deadlocks are easier to fix in SHIM because they are deterministic: on a given input, a SHIM program will either always deadlock or never do so.

```
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) {}
}
```

Figure 1: A SHIM program with exceptions

3. THE CELL PROCESSOR

Coherent shared memory multiprocessors, such as the Intel Core Duo, follow a conservative evolutionary path. Unfortunately, maintaining coherence costs time, energy, and silicon because the system must determine when data is being shared, and relaxed memory ordering models [1] makes reasoning about coherence difficult.

The Cell processor [15,17,23], the target of our compiler, instead 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 vector operations.

Our compiler uses multiple cores to provide task-level parallelism. Others address the Cell's vector-style data parallelism [8].

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. That our compiler relieves the programmer from having to program the Cell's memory flow controllers (DMA units) is a key benefit.

3.1 DMA and Alignment

The centerpiece of the Cell's communication system—and a major concern of our compiler—is the element interconnect bus (EIB): two pairs of counter-rotating rings [3,17], each 128 bits (16 bytes—a quadword) wide.

The width of the EIB leads the DMA units to operate on 128-bit-wide memory. Memory remains byte-addressed, but the 128-bit model puts substantial constraints on transfers because of the lack of byte-shifting circuitry [13, p. 61].

A DMA unit most naturally transfers quadwords. It can copy between 1 and 1024 quadwords (16K) per operation; source and destination addresses must be quadword-aligned.

A DMA unit can also transfer 1, 2, 4, or 8 bytes. The source and destination addresses must be aligned on the transfer width and have the same alignment within quadwords. For example, a 7-byte transfer requires three DMA operations, and transferring a byte from address 3 to address 5 requires a DMA to a buffer followed by a memory-to-memory move. To perform DMA operations, our compiler generates code that calls complex C macros that usually distill down to only a few machine instructions.

The alignment restrictions impose memory allocation constraints beyond the usual ones provided by C compilers, e.g., ensuring two regions have the same alignment within a quadword is difficult.

Our compiler produces C code suitable for the port of GCC to the SPE. We take advantage of a GCC extension that can place additional alignment constraints on types and variables. For example, a *struct* type or array variable can be constrained to start on a 16-byte boundary (e.g., to make it work with the DMA facility):

```
struct foo { int x, y; } __attribute__ ((aligned (16)));
int z[10] __attribute__ ((aligned (16)));
```

3.2 Mailboxes and Synchronization

For synchronization, our compiler generates code that uses the Cell's mailboxes: 32-bit FIFO queues for communication between the PPE and an SPE. Each SPE has two one-entry mailboxes for sending messages to the PPE and one four-entry queue for messages from the PPE [13, p. 101].

We use mailboxes for synchronization messages between the main program running on the PPE and tasks running on the SPEs. The SPE writing to an outbound mailbox causes an interrupt on the PPE, prompting it to read and empty the mailbox. In the other direction, the PPE writes to the SPE's inbound mailbox and can signal an interrupt on the SPE, but we just do a blocking read on the inbound SPE mailbox to wait for the next message.

All our communication is done using handshaking through the mailboxes; our protocol ensures the mailboxes do not overflow.

The Cell also provides signals: 32-bit registers whose bits can be set and read for synchronization; our code does not use them.

4. OUR COMPILER

We generate asymmetric code because of asymmetries in the Cell architecture and runtime environment. For example, the PPE supports pthreads but we do not know of a similar library for the SPEs. Also, mailboxes, the more flexible of the Cell's two synchronization mechanisms, work best between the PPE and an SPE. They can be used between SPEs, but are more awkward.

These considerations, along with our experience in implementing SHIM on shared-memory systems [7], led us to adopt a "computational acceleration" model [15] in which the SPEs run more time-critical processes and the PPE is responsible for the rest, including coordination among the SPEs. Communication in the code we generate takes place between the PPE and an SPE.

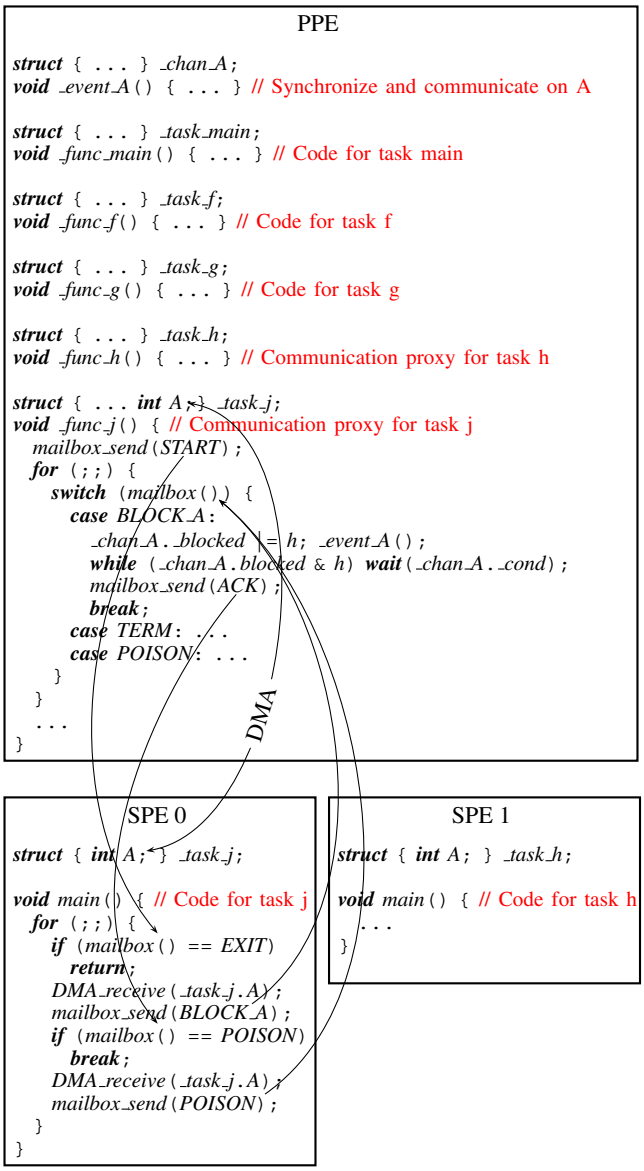


Figure 2: The structure of the code our compiler generates for the program in Figure 1. Each task becomes a function on the PPE; tasks that run on an SPE communicate with a PPE-resident proxy function using mailboxes and DMA.

Figure 2 shows the structure of the code we generate, here for the small example from Figure 1. We instructed our compiler to assign tasks *h* and *j* to two SPEs; all the others run on the PPE.

For PPE-resident tasks, our compiler generates almost the same pthreads-based code we presented in earlier work [7]. For each SPE-resident task, we generate SPE-specific code that communicates through mailboxes and DMA to a proxy function running on the PPE (e.g., *_func_j* in Figure 2). The SPE functions, shown at the bottom of Figure 2, translate communication from the SPE code to the PPE-resident pthreads environment.

4.1 Code for the PPE

The C code we generate for the PPE uses the pthreads library to emulate concurrency much like we did for our shared-memory compiler [7]. Each task and each channel has its own shared data

```
int foo(int a, int &b, chan uint8 cin, chan uint8 &cout) {
  next cout = a; next cout = b; next cout = next cin;
  return '\n';
}

struct {
  pthread_t _thread;
  pthread_mutex_t _mutex;
  pthread_cond_t _cond;
  enum _state { _STOPPED, _RUNNING, _POISONED } _state;
  unsigned int _attached_children;
  unsigned int _dying_children;
  int *b;
  int *__return_var;
  struct {
    struct {
      unsigned char cout;
      int b;
      int __return_var;
    } _byref;
    unsigned char cin;
    int a;
  } _args __attribute__((aligned (16)));
} _foo;
```

```
struct {
  pthread_mutex_t _mutex;
  pthread_cond_t _cond;
  unsigned int _connected;
  unsigned int _blocked;
  unsigned int _poisoned;
  unsigned int _dying;
  unsigned char *foo;
  unsigned char *main_1;
  unsigned char *main;
} _cin;
```

Figure 3: Shared data for the *foo* task and *cin* channel.

structure that includes a lock used to guarantee access to it is atomic and a condition variable for notifying other threads of state changes (Figure 3). Each of these resides in main (PPE) memory and are manipulated mostly by the PPE code.

For each SHIM function, our compiler generates a C function that runs in its own thread. Our code starts a thread for each SHIM function when the program starts to minimize the overhead of creating and terminating threads. With the exception of the thread for the entry point, each thread immediately blocks until its parent calls it.

For each channel, we generate an *event* function responsible for managing synchronization and communication on the channel (e.g., *_event_A* at the top of Figure 2). For speed, our compiler "hard-wires" the logic of each *event* function because a SHIM program's structure is known at compile time. A generic function controlled by channel-specific data would be more compact but slower.

4.2 Code for the SPEs

For each SHIM function that will execute on an SPE, we generate a C function and compile it with the standard port of GCC to the SPEs. Again, most of SHIM is translated mechanically into C; code for communication and synchronization is the challenge.

Our strategy is to place most of the control burden on the PPE and use the SPEs to offload performance-critical tasks. This simplifies code generation by removing the need for inter-SPE synchronization; we only need an SPE-PPE mechanism.

Using command-line arguments, the user specifies one or more "leaf" functions to run on the SPEs, such as tasks *h* and *j* in Figure 2. Such functions may communicate on channels, but may not start other functions in parallel or call functions that communicate. However, a leaf function may call other functions that do not communicate or invoke functions in parallel, i.e., those that behave like standard C functions. This restriction saves us from creating a mechanism for starting tasks from an SPE.

The pthreads synchronization mechanisms (mutexes, condition variables) our code uses do not work across the PPE/SPE boundary.[1] Instead, for each function destined for an SPE, we synthesize

[1]IBM's "Example" library [14] does provide cross-processor mutexes, but blocking operations never yield to the thread scheduler.

a proxy function on the PPE that acts as a proxy for the function on the SPE that does the actual work (*_func_j* and *_func_h* in Figure 2). Each proxy translates between pthreads events on the PPE and mailbox events from the SPE.

Passing arguments to an SPE task turns out to be awkward because of DMA-imposed alignment constraints. Our solution requires two copies: a DMA transfer from the PPE followed by word-by-word copying into local variables, which allows the compiler to optimize their access. This is one of the few cases where a C compiler is a disadvantage over generating assembly.

Channel communication is done through mailbox messages for synchronization and DMA for data transfer (Figure 2). It starts when the SPE task sends a BLOCK message to the PPE for a particular channel. This prompts the PPE proxy to signal it is blocked on that channel. When the *event* function on the PPE releases the channel (i.e., when all connected tasks have rendezvoused), the PPE sends an ACK message to the SPE, which prompts it to start a DMA transfer to copy the data for the channel from the argument *struct* on the PPE to a matching *struct* on the SPE. There is no danger of this data being overwritten because only the *event* function on the PPE writes the *struct*, and that will only happen after the task is again blocked on the channel, which will not happen until the SPE task requests it, which will only happen after the DMA is complete.

A task may become "poisoned" when it attempts a rendezvous and another task in the same scope has thrown an exception. The *event* function in the PPE code handles the logic for propagating exception poison; the PPE proxy code is responsible for informing the SPE task it has been poisoned.

The SPE code may send two other messages. TERM is the simpler: the SPE sends this when it has terminated, and the PPE proxy jumps to its own *_terminate* handler, which informs its parent that it has terminated. The other message is POISON, which the SPE code sends when it throws an exception. After this, it sends another word that indicates the specific exception. Based on this word, the proxy marks itself and all its callers in the scope of the exception as poisoned, then jumps to the *_poisoned* label, which also handles the case where the task has been poisoned by a channel.

5. COLLECTING PERFORMANCE DATA

While tuning our compiler and applications, we found we needed pictures of the temporal behavior of our programs. While speeding up any part of a sequential program is beneficial, improving a parallel program's performance requires speeding computation along a critical path—any other improvement is hidden.

To collect the data we wanted, we added a facility to our compiler that collects the times at which communication events begin and end. For this, we use the SPE's "decrementer"—a high-speed (about 80 MHz) 32-bit software-controlled countdown timer. Our compiler can add code that reads this timer and stores the starting and stopping times of each communication action, i.e., periods when the SPE is blocked waiting for synchronization. We fill a small buffer in the SPE's local store, then dump the event timestamps into a text file when the program terminates. Our goal is to be as unintrusive; each sample event consists of testing whether the buffer is full, reading the timer, writing into an array, and incrementing a destination pointer.

To understand the interaction among SPEs, we wanted global time stamps, so we include code to synchronize the decrementers. Although the SPEs' decrementers run off a common clock, their absolute values are set by software and not generally synchronized.

Our synchronization code measures round-trip communication time and uses it to synchronize the clocks on the SPEs. We assign one SPE to be the master, then synchronize all the other SPEs' clocks to it. The master first establishes communication with the slave (i.e., waits for the slave to start), then sends a message to the slave through its mailbox, which immediately sends it back. The master measures the time this took—the round-trip time. Finally, the master sends the current value of its clock plus half the round-trip time to the slave, which sets its clock to that value.

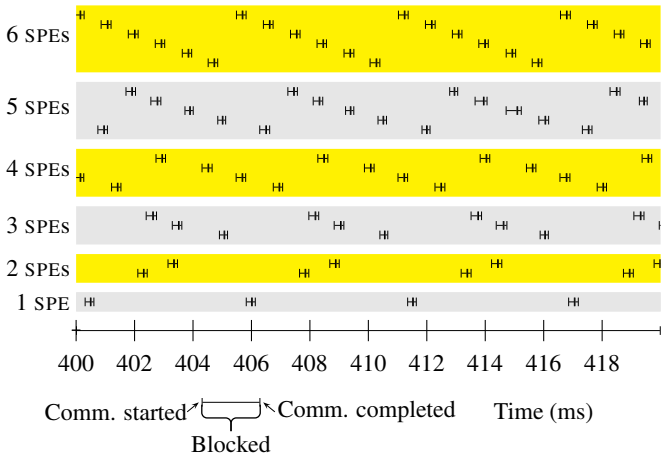


Figure 4: Temporal behavior of the FFT for various SPEs

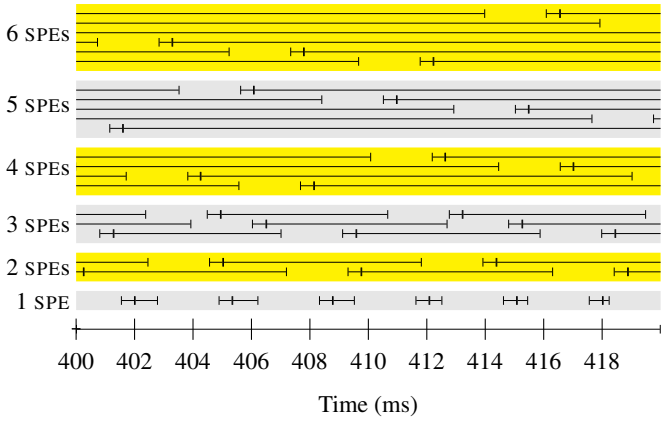


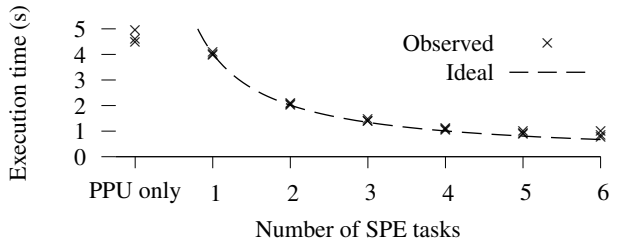
Figure 5: Temporal behavior of the JPEG decoder

Figures 4 and 5 shows data we obtained with this mechanism. Time runs from left to right, and each line segment denotes the time that one SPE is either blocked or communicating; empty spaces between horizontal lines indicate time an SPE is doing useful work. The vertical position of each line indicates the SPE number.

6. EXPERIMENTAL RESULTS

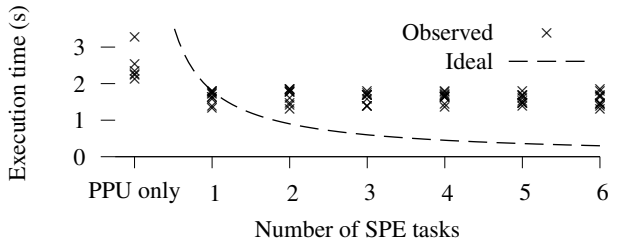
To evaluate our compiler, we used it to compile a pair of applications and ran them on a Sony Playstation 3 running Fedora Core 7 with Linux kernel 2.6.23 and the IBM SDK version 3.0.

The Sony Playstation 3 is a Cell-based machine with 256 MB of memory, a single Cell with one SPE disabled to improve yield, and peripherals including an Ethernet interface and a hard drive. While the PS3 platform is open enough to boot an operating system such as Linux, it does not allow full access to the hardware. Instead, guest operating systems run under a hypervisor that limits access to the hardware such as the disk, only part of which is visible to Linux. The hypervisor on the PS3 also reserves one of the SPEs for security tasks, leaving six available to our programs.



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

Figure 6: Running time for the FFT on varying SPEs



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

Figure 7: Running time for the JPEG decoder on varying SPEs

We compiled the generated C code with GCC 4.1.2 for the PPE and 4.1.1 for the SPE code, both optimized with `-O`.

Figure 6 shows execution times for an FFT that takes an audio file, divides it into 1024-sample blocks, performs a fixed-point (4.28) FFT on each block, follows it by an inverse FFT, and writes it out to a file. A PPE-based reader tasks distributes 8 1024-sample blocks to the SPE tasks in a round-robin order; a writer task collects them in order and writes them out to a file. We communicate 8 blocks instead of the 16 we used earlier [7] to accommodate the SPEs' local store. We ran this on a 20 MB stereo audio file with 16-bit samples. The "PPU only" code is from our earlier compiler [7].

Figure 4 illustrates why we observe a near-ideal speedup for the FFT on six SPEs. Roughly half the time all six are doing useful work; otherwise one is blocked communicating, giving a speed-up of about $11/2 = 5.5$, close to the 5.3 we observed (Figure 6).

Each horizontal line in Figure 4 represents two events: an FFT task on an SPE reads a block, processes it, sends it, and then repeats the process; the read immediately follow the write. The figure also shows that the processes spend more time blocking waiting to write than they do to read, suggesting the task that reassembles data from the FFT tasks is slower than the one that parcels it out.

We also compiled and ran a JPEG decoder, similar to our earlier work [7]. Figure 7 shows the execution times we observed, which do not exhibit the same speedup as the FFT and are much more varied. Figure 5 explains why: for these runs, the SPEs are spending most of their time waiting for data. For this sample, only at one point the 3-SPE case is more than one SPE active at any time.

Figure 5 tells us the SPEs are usually waiting for data to arrive. Each line segment is actually two parts: sending processed data (left), and receiving unprocessed data. This is not surprising; while JPEG data is composed of independent blocks, the data itself is Huffman encoded, meaning it requires the data to be uncompressed before block boundaries can be identified.

The performance figures we report are for carefully chosen problem sizes. Start-up overhead is larger for smaller problems sizes, leading to poorer results; the data for larger problem sizes does not fit into the PS3's 256 MB of main memory, necessitating disk access that quickly becomes the bottleneck. For large data sets, our performance degrades to just disk I/O bandwidth, suggesting the PS3 is not ideally suited to large scientific computing tasks.

7. RELATED WORK

Other groups that have produced compilers for the Cell start from models very different from SHIM and address different problems.

Eichenberger et al.'s compiler [8, 9] takes a traditional approach by starting with C code with OpenMP annotations [21] and generates code for the Cell. They consider low-level aspects of code generation: vectorizing scalar, array-based code; hiding branch latency; and ensuring needed data alignment. They implement the OpenMP model: programmers provide hints about parallelizable loops, then the compiler breaks these into separate tasks and distributes them to the SPEs. It presents a shared memory model, which their runtime system emulates with explicit DMA transfers.

OpenMP is a much different programming model than SHIM: it assumes shared memory and focuses on parallelizing loops with array access. SHIM, by contrast, is a stream-based language with explicit communication. Adding OpenMP-like constructs to improve SHIM's array performance would be a nice complement.

Adopting a more SHIM-like message passing approach, Ohara et al.'s [20] preprocessor takes C programs written using the standard message passing interface (MPI) API [19], determines a static task graph, clusters and schedules this graph, and finally regenerates the program to use Cell-specific API calls for communication.

Semantically, the MPI model is similar to SHIM but does not guarantee scheduling independence. The big difference is that the preprocessor of Ohara et al. does not enforce the programming style; it would be easy to write a misbehaving program. The SHIM compiler catches a host of bugs including deadlock [27].

Fatahalian et al.'s Sequoia [10] is most closely related to our work. Like us, they compile a high-level concurrent language to the Cell processor (and other architectures) with the goal of simplifying the development process.

Their underlying computational model differs substantially from SHIM's, however. While also explicitly parallel, it is based on stateless procedures that only receive data when they start and only transmit it when they terminate. This model, similar to the one in Cilk [4], is designed for divide-and-conquer algorithms that partition large datasets (typically arrays) into pieces, work on each piece independently, then merge the results. While our example applications also behave this way, other SHIM programs do not.

While the low-level compilation challenges of the Cell are fairly conventional, higher-level issues are less obvious. Because the processor is young and idiosyncratic, there is still work to be done in choosing strategies for structuring large programs. For example, Petrini et al. [22] observe a high performance implementation of a three-dimensional neutron transport algorithm requires a careful balance among vector parallelism in the SPEs, the effect of their pipelines, balancing and scheduling DMA operations, and coordinating multiple SPEs. Saidani et al. [24] change DMA transfer sizes to improve the performance of an image processing algorithm. Gedik et al. [11] optimize distributed sorting algorithms on the Cell by careful vectorization and communication. They note main memory bandwidth becomes the bottleneck on large datasets since the inter-SPE bandwidth is so high. Our compiler only provides higher-level data communication and synchronization facilities.

Chow et al. [5] discuss coding a large FFT on the Cell. They suggest putting the control of the application on the PPE, then offloading computationally intensive code to the SPEs and adapting it to work with the SPEs' vector capabilities. We adopt a similar philosophy in the code generated by our compiler.

They target their application at a 128 MB dataset—too large to fit in on-chip memory, so much of their design concentrates on orchestrating data movement among off-chip memory, the PPE's cache, and the SPEs' local stores. They divide the FFT into three stages and synchronize the SPEs using mailboxes on stage boundaries.

8. CONCLUSIONS

We described a compiler for the SHIM concurrent language that generates code for the Cell processor. While not an aggressive optimizing compiler, it removes much of the drudgery in programming the Cell in C, which requires extensive library calls for starting threads, careful memory alignment of data if it is to be transferred between processors, and many other nuisances.

The SHIM language presents a scheduling-independent model to the programmer, i.e., relative task execution rates never affects the function computed by the program. This, too, greatly simplifies the programming task because there is no danger of introducing races or other nondeterministic behavior.

Unfortunately, our compiler does not solve a main challenge of parallel programming: creating well-balanced parallel algorithms. For example, the sequential portion of our FFT was able to keep six SPEs fed, leading to near-ideal speedups; the sequential portion of the JPEG decoder was substantial and became the bottleneck.

Our compiler does help to identify bottlenecks: it provides a mechanism for capturing precise timing traces using the Cell's precision timers. This gives a precise summary of when and how long each SPE is blocked waiting for communication, which can illustrate poorly balanced computational loads.

The Cell processor is an intriguing architecture that is representative of architectures we expect to find in many future embedded systems. While it has many idiosyncrasies, our work shows that it is possible to map a higher-level parallel programming model onto it and obtain reasonable performance.

In the future, we plan extensions to SHIM that make it easier to express instruction-level parallelism. While SHIM is currently effective for expressing task-level parallelism, it is weak at expressing the SIMD-like vector operations supported on both types of cores in the Cell. At the moment, we rely on the C compiler to exploit these features, but language extensions should greatly improve designers' control over these features.

9. REFERENCES

[1] S. V. Adve et al. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[2] V. Agarwal et al. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Intl. Symp. Computer Architecture (ISCA)*, pages 248–259, June 2000.

[3] T. W. Ainsworth and T. M. Pinkston. Characterizing the Cell EIB on-chip network. *IEEE Micro*, 27(5):6–14, Sept. 2007.

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

[5] A. C. Chow et al. A programming example: Large FFT on the Cell Broadband Engine. In *Global Signal Processing Expo (GSPx)*, Santa Clara, CA, Oct. 2005. (from IBM)

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

[7] S. A. Edwards, N. Vasudevan, and O. Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In *Proc. Design, Automation, and Test in Europe (DATE)*, pages 1498–1503, Munich, Germany, Mar. 2008.

[8] A. E. Eichenberger et al. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. *IBM Sys. J.*, 45(1):59–84, 2006.

[9] A. E. Eichenberger et al. Optimizing compiler for the CELL processor. In *Par. Arch. and Compilation Techniques (PACT)*, pages 161–172, Saint Louis, MO, Sept. 2005.

[10] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Supercomputing (SC)*, Tampa, FL, 2006. Article 83.

[11] B. Gedik et al. CellSort: High performance sorting on the Cell processor. In *Very Large Data Bases (VLDB)*, pp. 1286–1297, Vienna, Austria, Sept. 2007.

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

[13] IBM. *Cell Broadband Engine Architecture v1.02*, Oct. 2007.

[14] IBM. *Example Library API Reference v3.0*, Sept. 2007.

[15] J. A. Kahle et al. Introduction to the Cell multiprocessor. *IBM J. of R&D*, 49(4/5):589–604, July/Sep. 2005.

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

[17] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, May-June 2006.

[18] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9):1235–1245, Sept. 1987.

[19] The Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 1995. Version 1.1.

[20] M. Ohara, H. Inoue, Y.Sohda, H. Komatsu, and T. Nakatani. MPI microtask for programming the Cell Broadband Engine processor. *IBM Systems Journal*, 45(1):85–102, 2006.

[21] OpenMP Arch. Review Board, www.openmp.org. *OpenMP C and C++ Application Program Interface*, 2002. Ver. 2.0.

[22] F. Petrini et al. Multicore surprises: Lessons learned from optimizing Sweep3D on the Cell Broadband Engine. In *Intl. Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, Long Beach, CA, Mar. 2007.

[23] D. Pham et al. The design and implementation of a first-generation Cell processor. In *Solid-State Cir. Conf. (ISSCC)*, v. 1, pp. 184–185, San Francisco, CA, Feb. 2005.

[24] T. Saidani, S. Piskorski, L. Lacassagne, and S. Bouaziz. Parallelization schemes for memory optimization on the Cell processor: A case study of image processing algorithm. In *Workshop on Memory Performance: Dealing with Applications, Systems and Architecture (MEDEA)*, pages 9–16, Brastov, Romania, Sept. 2007.

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

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

[27] N. Vasudevan and S. A. Edwards. Static deadlock detection for the SHIM concurrent language. In *Formal Methods and Models for Codesign*, Anaheim, CA, June 2008.