Communicating Process Architectures 2006
Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)
IOS Press, 2006

A Cell Transterpreter

Damian J. DIMMICH, Christian L. JACOBSEN, Matthew C. JADUD

Computing Laboratory, University of Kent,
Canterbury, CT2 7NF, UK.

{djd20 , clj3 , mcj4} @kent.ac.uk

Abstract. The Cell Broadband Engine is a hybrid processor which consists of a PowerPC core and eight vector co-processors on a single die. Its unique design poses a number of language design and implementation challenges. To begin exploring these challenges, we have ported the Transterpreter to the Cell Broadband Engine. The Transterpreter is a small, portable runtime for concurrent languages and can be used as a platform for experimenting with language concepts. This paper describes a preliminary attempt at porting the Transterpreter runtime to the Cell Broadband Engine and explores ways to program it using a concurrent language.

Keywords. CELL processor, Transterpreter, Portable run-time environments

Introduction

Multi-core processors are becoming commonplace in desktop computers, laptops and games consoles [1,2,3]. Traditionally programming such concurrent systems has been considered difficult [4,5]. We believe that creating and designing software that makes use of concurrency provided by hardware can be easy, given a language and runtime that provide support for concurrency. With a compiler that can check for certain errors in concurrent code, and a runtime that gracefully reports error conditions surrounding program deadlock, developing such software can become simpler.

The Sony PlayStation III [6], a consumer games console which will be released at the end of 2006, is an example of a readily available and affordable multi-core system. At its heart, the PlayStation III will be powered by the Cell Broadband Engine [7], commonly referred to as the Cell processor. The Cell, as seen in Figure 1, has a complicated architecture consisting of a PowerPC (PPC) core surrounded by eight vector processors, called Synergistic Processing Units [8] (SPUs). These nine processors are connected by a high-speed bus that provides fast inter-processor communication and access to system memory.

The Transterpreter [9,10] is a small and highly portable runtime for concurrent language research and is used as the platform for the implementation described. The Cell edition of the Transterpreter aims to address architectural issues that need to be overcome for the runtime to function in a useful manner. occam-π, a language supported on the Transterpreter, has built in semantics for concurrency, and provides a compiler which supports the programmer in developing safe concurrent software [11]. occam-π provides language level facilities for safe interprocess communication and synchronisation, and is derived from a formal model of concurrency that can be used to reason about programs [12,13].

The long-term goals of this project are to address the difficulties of programming the Cell processor and other concurrent systems which require a more involved design, such as a pipeline of processes or mulitple instruction multiple data (MIMD) type problems, and cannot be effectivley parallelised through the use of preprocessor directives and loop level concurrency. This paper describes the implementation of a prototype concurrent runtime for the Cell processor, a first attempt at reaching our goals.

We begin this paper with an overview of the Cell processor's architecture, and the steps taken to port the Transterpreter to the Cell. We then present an overview of what programming the Cell using occam-π could involve, and close with a discussion of future work.

1. An Overview of the Cell Broadband Engine

The Cell Broadband Engine consists of a PowerPC core that is connected to eight vector processing units which are connected via a high speed bus. This architecture provides a number of challenges for language designers and programmers. While it was not possible to purchase a Cell processor at this time of writing, a cycle accurate simulator for the Cell was available [14]. The simulator, available for Linux, lets programmers begin developing software for the Cell Broadband Engine. What follows presents background information on the Cell architecture which should help clarify some of the implementation details given later. This section closes with an overview of the challenges that this architecture presents and discusses IBM's attempt at addressing them.

1.1. PowerPC Core (PPC)

The PowerPC core was designed to be binary compatible with existing PowerPC based processors, allowing the execution of pre-existing binaries on the Cell's PPC core without modification. The core has built-in hardware support for simultaneous multithreading, allowing two threads to execute in parallel on the processor. An AltiVec [15] Single-Instruction-Multiple-Data (SIMD) vector-processing unit is also present.

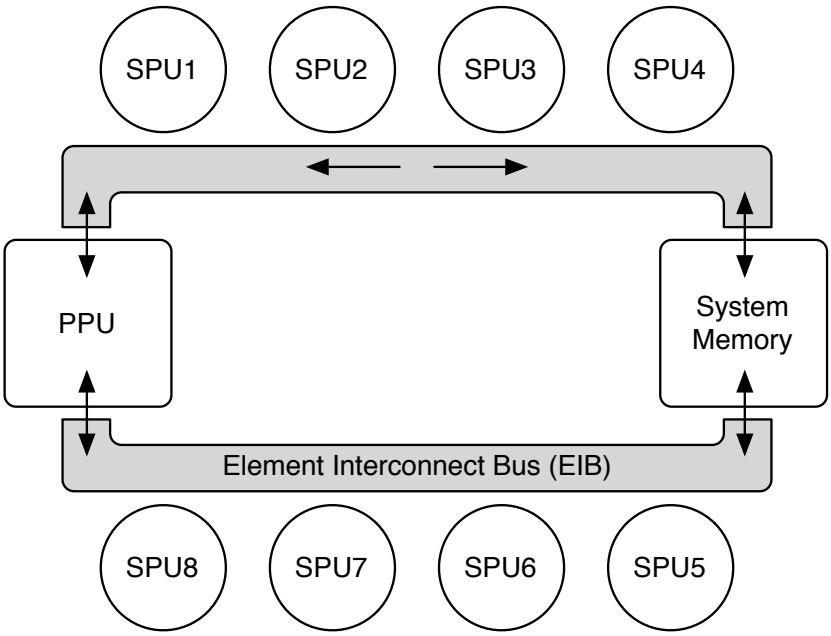

Figure 1. A diagram of the Cell BE.

1.2. Synergistic Processing Units (SPU)

The SPU processors are dedicated vector processing units. Each SPU is equipped with a 256KB local store for program and data, and each unit has a dedicated memory controller attached to it. The memory controller, and hence memory access, is programmed explicitly from the SPU to manage data flow between system memory and its local store. The memory

controllers are able to move chunks of data up to 16KB in size to and from the SPU units' local stores without interrupting computation.

The memory controllers also coordinate most of the inter-processor synchronisation. Synchronisation and communication are achieved by reading from and writing to special-purpose memory-mapped registers designated for this task. Each SPU is equipped with a set of three 32-bit mailboxes (registers) for communication/synchronisation with the PPC. Two are outbound, blocking mailboxes, one of which interrupts the PPC when a message is sent. The third, inbound mailbox, is for receiving messages from the PPC. Each SPU is also equipped with two inbound 32-bit registers, called `SigNotify1` and `SigNotify2`, which any SPU or PPC can write to, with either overwriting or blocking behaviours.

1.3. The Cell's Element Interconnect Bus

A key component of the Cell Processor is the high-speed Element Interconnect Bus (EIB) through which all of the processing units and main memory are connected. The EIB is an on-chip bus which allows all the processing units to communicate with each other directly, without requiring access to main memory. A diagram of the Cell processor and how the Cell interconnects all of the elements can be seen in Figure 1 on the facing page.

1.4. The Cell's Challenges

The Cell processor contains two different processor types, and nine independent processor segments. This means that programs wishing to exploit all processors have to be effectively written as two or more separate programs, one of which is compiled for and executes on the PPC, and another which is compiled for and executes on the SPU. These separate programs have to be able to synchronise and share data. This and the need to manage the SPU memories explicitly make programming for the Cell considerably more difficult than programming for traditional processors.

IBM, being aware of these difficulties has chosen to address them by developing an auto-parallelising compiler [16] for the Cell processor. It attempts to address these issues by relying on the concurrency inherent in loops, and by using preprocessor directives [17] that instruct the compiler about transformations that are safe in order to create parallel code. Such auto-parallelisation provides a fast mechanism for making use of the additional processor power available on a concurrent system without the need for rewriting existing code. Unfortunately not all code can be modified or annotated easily to gain performance advantages from auto-parallelising compilers. An alternative method to automatic parallelisation could provide the ability to make use of the Cell without requiring expert programming.

2. The Transterpreter on the Cell Broadband Engine

The Transterpreter is a highly portable interpreter for concurrent languages. It provides a means for running a concurrent language on a new platform in a short amount of time, circumventing the need to write a new compiler backend for a given language. With the port of the Transterpreter to the Cell, we hope to explore how we can use occam-π, a language designed with concurrency in mind, in the context of the Cell processor and gain understanding of what would be required in order to port the language to the Cell processor.

The core of the Transterpreter is portable across platforms because it has no external dependencies, and it builds using any ANSI compliant C compiler. For the Transterpreter runtime to be useful on a given architecture, a platform specific wrapper needs to be written which provides an interface to the underlying hardware. In the case of the Cell two separate wrappers where needed, one for the PPC core, and one for the SPUs.

Porting the Transterpreter to the Cell required careful thought regarding occam-π channels and scheduling. One of the things that had to be taken into account when writing the wrappers was the mapping of occam-π channels to the Cell's communication hardware. This was difficult because the underlying hardware does not map directly to the semantics of CSP/occam-π channels. Furthermore, the Transterpreter's scheduler needed to be extended to support the handling of interrupts generated by the Cell.

2.1. Program Distribution

The Transterpreter for the Cell consists of a PPC executable which is programmed to load and start SPU Transterpreter executables. Once the PPC and the SPUs are all running, the program bytecode designated for execution is loaded into main memory. A pointer to the program bytecode is passed to all the SPUs which then copy the program bytecode into their local stores from system memory. Once the copy is complete all the Transterpreters begin executing the bytecode. Currently all Transterpreter instances receive the same copy of the bytecode. In order for program flow to differ on processors, a program must query the Transterpreter about it's location in order to determine what to do. A program's location can be determined by the unique CPU ID, a number from 0 to 9, that each Transterpreter instance gets assigned at startup.

2.2. Inter-Processor Communication

occam-π channels are unidirectional, blocking, and point-to-point. The individual SPUs of a Cell processor are not so limited in their communications; therefore, both the compiler and the wrappers must provide support for making our mapping from occam-π to the Cell hardware consistent and safe. The blocking nature of channel communications provides explicit synchronisation points in a program. While the compiler provides checks for correct directional usage of channels when compiling, the Transterpreter wrappers must ensure such that that channel communications between processors are blocking and unbuffered.

2.2.1. SPU to PPC Communication

The SPU-to-PPC mailbox registers are word-sized (32-bit), unidirectional non-overwriting buffers. When empty, a mailbox can be written to, and execution can continue without waiting for the mailbox to be read. When a mailbox is read, it is emptied automatically. When a mailbox is full, the writing process will stall until the previous message has been read. The SPU can receive interrupts when one of its mailboxes is read from, or written to.

The mailbox registers are used to implement channel communications in occam-π between the PPC and the SPU. In order to preserve the channel semantics of occam-π, a writing process is taken off the run queue and set to wait for the "mailbox outbound read" interrupt to occur. The communication only completes when the mailbox is read by the PPC. The SPU is able to quickly pass multi-word messages by continuously writing to the mailbox while the PPC continuously reads. The PPC does not receive interrupts when its outbound message is read and must poll the mailbox to check if it has been read.

2.2.2. Inter-SPU Communication

For SPU-to-SPU communications, two inbound registers `SigNotify1` and `SigNotify2`, are provided on each SPU. The registers are programatically configured by the Transterpreter to be non-overwriting and, like the mailbox registers, can only be cleared by a read. The Transterpreter provides facilities for sending both large messages and short, word-size messages between SPUs.

On Transterpreter startup, space is reserved in main memory for the sending of large messages. Eight 16KB chunks of memory are allocated for each SPU to receive data in.

Each SPU then receives a list of pointers to the memory locations that they can write to. For example, SPU 2 will receive pointers to the second 16KB chunk of memory of each SPU's receiving memory. This ensures that when two SPUs are sending data to a third SPU, no portion of memory is overwritten accidentally.

When a write to another SPU is initiated, the data to be sent is copied into the appropriate 16KB chunk of memory. Once the copy completes, the writer SPU puts its CPU ID into `SigNotify1` on the destination SPU. The writer process is then moved to the interrupt queue in the scheduler and waits for confirmation of the read.

Once `SigNotify1` has been written to on the reader SPU, an interrupt is generated on the it, informing the SPU about the write. If a process is waiting on the interrupt queue awaiting data, it copies the message in main memory into its local store, using the 'read-only' pointer that it was provided with at startup. Once the copy is completed the reader SPU acknowledges completion of the read by putting a flag value into `SigNotify1` on the writer SPU to confirm that the read has been completed. At this point the reading SPU process can be taken off the interrupt queue and can continue execution. The writer SPU process which had been waiting on the interrupt queue checks that it has received a read confirmation from the reader SPU, and can continue executing once it is rescheduled.

Alternatively, for short messages, or synchronisation it is possible to send word-sized messages between SPUs. This capability is useful since it allows for communication and synchronisation without taxing the memory bus. In this case the `SigNotify2` register is used for sending data and `SigNotify1` for the sender's CPU ID. In order to determine the length and type of message being sent, the program must be written such that each SPU may only send one message at a time to another SPU. Furthermore, the reader and the writer have to agree on the size of the message at compile-time. These factors determine if a memory copy or the `SigNotify2` register is used for sending data.

2.3. Scheduling

The Transterpreter uses a co-operative scheduler with three queues: the run queue, the timer queue and the interrupt queue. Process rescheduling only occurs at well defined points during execution. This implies that the interrupt queue is only checked when the scheduler is active. If interrupts are ready, processes waiting on them are moved from the interrupt queue to the back of the run queue.

This behaviour strongly encourages programmers to make use of occam-π's concurrency features so as not have a processor stalling whenever a process needs to wait on an interrupt. occam-π processes have a very low overhead in terms of memory usage and context switching. This allows a programmer to develop programs with many concurrent processes executing that will make use of the processor while some processes are waiting on external communication to complete.

Should a programmer wish to write programs using a more sequential paradigm, buffer processes could be used to handle communications. Figure 2 illustrates how a writing buffer process can be used when communicating with another processor. This way, only the buffer process stalls while waiting for communication to complete, allowing other processes to continue executing. Similarly, a dedicated reading buffer process can ensure that data is read in as soon as it is available reducing potential stalls in the network and keeping all the processors busy. This can be made particularly effective by running the buffer processes at higher priority than other processes.

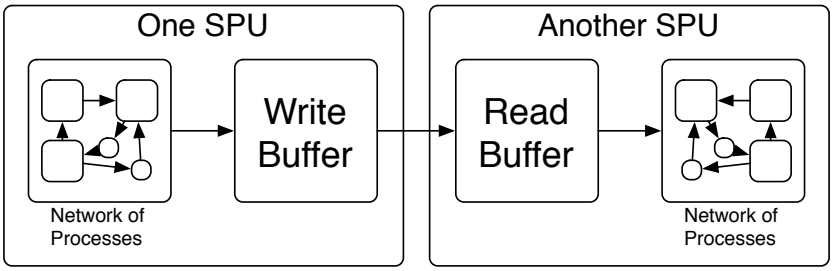

Figure 2. The read and write buffers allow computation to continue with less interruption.

3. Programming the Cell Using the Transterpreter

The Transterpreters that are executing on the SPUs are each assigned a unique CPU ID. A native function call mechanism in occam-π allows the programmer to call C functions that are a part of the Transterpreter. Using the native call `TVM.get.cpu.id`, the program can determine on which processor it is executing. A CPU ID value of 0 is returned if the bytecode is running on the PPC, and a value between 1 and 8 if it is on one of the SPUs. An example of how an occam-π startup process on the Cell could be written is shown in Listing 1.

```
PROC startup(CHAN BYTE kyb, err, src)
  INT id:
  SEQ
    TVM.get.cpu.id(id) --check where we are running
    IF
      id = 0
        ... -- execute PPC code.
      id > 0
        ... -- execute SPU code.
:
```

Listing 1. An example of a startup process on the Cell Transterpreter.

In order to send and receive messages between processors, the native functions `TVM.read.mbox` and `TVM.write.mbox` are provided. These native functions behave similarly to occam-π channels in that they block until the communication has completed. An example of their use is shown below in a program where nine Transterpreters are running concurrently and are connected in a ring. All the processes in the pipeline do is increment a value as it propagates through. The result of the incrementing is printed each time it comes back to the process that originated the value.

The process `run.on.spu` in Listing 2 reads values from the previous processor in the pipeline. The value is then incremented and sent on to the next processor. Because of the "\" - the modulo operator, when the process is running on the processor with CPU ID 8, it sends the value back to the PPC who's CPU ID is 0.

```
VAL INT stop IS 95:
PROC run.on.spu(VAL INT cpuid)
  INITIAL INT value IS 0:
  INT status:
  WHILE value < stop
    SEQ
      TVM.read.mbox((cpuid - 1), value, status)
      value := value + 1
      TVM.write.mbox((cpuid + 1) \ 8, value)
:
```

Listing 2. A process running on an SPU.

The process `run.on.ppu` in Listing 3 is executed only on the PPC core. The process header denotes that a **`CHAN BYTE`** must be passed to it as a parameter. This is a channel of type **`BYTE`**, the equivalent of a `char` in C. In this program it is connected to the screen channel that is used as a method of output. The "!" is used to denote a write to a channel, where the RHS contains the value to be written, and the LHS the name of the channel to write to. It starts propagating a value down the pipeline by writing to the first SPU. It waits for the value to complete going through the pipeline and it outputs the modified value, followed by a return character by writing to the `scr` channel.

```
PROC run.on.ppc(CHAN BYTE scr!)
  INITIAL INT value IS 65:
  INT status:
  WHILE value < stop
    SEQ
      TVM.write.mbox(1, value)
      TVM.read.mbox(8, value, status)
      scr ! (BYTE value)
      scr ! '*n'
:
```

Listing 3. The process which runs on the PPC and outputs data to the screen.

Listing 4 shows the startup process, `startup`. This gets the CPU ID using the `TVM.get.cpu.id` function and runs the appropriate process depending on the value of `cpuid`. In this process, the `kyb`, `scr` and `err` channels are obtained 'magically' by the starting process much like command line parameters are obtained in C's `main` function. When the `run.on.ppu` process is started, the `scr` channel is passed to it as a parameter so that it can output to the screen. The `run.on.spu` process receives the CPU ID as a parameter.

```
PROC startup(CHAN BYTE kyb?, scr!, err!)
  INT cpuid:
  SEQ
    TVM.get.cpu.id(cpuid)
    IF
      cpuid = 0
        run.on.ppu(scr!)
      cpuid > 0
        run.on.spu(cpuid)
:
```

Listing 4. The startup process which starts the correct process depending where the code is running.

In future, a library can be created which will wrap around the native `TVM` calls to provide a channel interface for communicating between processes. This is desirable because it reflects the more common occam-π programming methodology, where processes communicate through channels to achieve synchronisation and send data.

4. Future Work

In the simplest case there are obvious improvements that can be made to the Transterpreter for running occam-π programs on the Cell. The most obvious are infrastructure modifications that will allow for using the channel abstraction for communication between processors. Further improvements could leverage the vector processing capabilities of the Cell BE architecture more effectively. Finally, we would like to explore code generation possibilities using the Transterpreter.

4.1. Infrastructure

The current implementation of the Transterpreter on the Cell does not provide a means for channel communications to occur between processors. Currently, sending data using the `TVM.write.mbox` in parallel would result in unpredictable behaviour. Direct support for channel communications would allow compile-time checking which could ensure a degree of safety.

Furthermore a method for abstracting the channel connections between processors, akin to pony's [18] virtual channels would allowing for multiple channels to exist between processors. This would enable a much more flexible mode of programming by multiplexing channels automatically.

4.2. Vector processing

While C does not have good concurrency primitives, some implementations have extended the language to include vector processing functionality. A C-based vector processing library could be embedded in the SPU wrappers for the Transterpreter, and an interface to it could be created using the occam-π extension for SWIG [19], a foreign-function interface generator.

The occam-π language could also be extended to support vector primitives using the languages' support for operator overloading [20]. To support vector primitives at a bytecode level, a compiler that provided a means for easily adding new syntax would be needed. The Transterpreter bytecode would need to be extended and the runtime modified accordingly to allow the updated bytecode to function.

4.3. Code Generation

While being able to run occam-π and the Transterpreter on a platform like the Cell is interesting [21], it is impractical. Even though the runtime has a small memory footprint in comparison with other virtual machines, the limited amount of local store available on the SPUs becomes even smaller when it needs to be shared by bytecode and the Transterpreter. Additionally, the current implementation of the Transterpreter replicates the same bytecode to each processor, meaning that unused code is being carried around, wasting further resources.

Furthermore, the overhead of running a bytecode interpreter is particularly large since the SPU is only capable of 128-bit-sized loads and stores, and a penalty is paid on all scalar operations due to the masking and unmasking necessary for processing 32-bit values.

Because of the high memory consumption, and the overhead caused by bytecode interpretation, occam-π and the Transterpreter become unattractive to developers who need the specialised processing power that the Cell offers. In order to address these issues, we have begun exploring the idea of generating native code from occam-π [22] using parts of the Transterpreter runtime and gcc [23]. Such a solution will allow us to combine the speed associated with C, and the safe concurrency that occam-π have to offer.

A manner of either automatically inferring, or specifying through annotation, which parts of the code are required on which processor would allow for dead code elimination and hence, smaller binaries on the SPUs.

The current implementation of the Transterpreter for the Cell provides a basis for code generation for the Cell. It has shown that a language like occam-π is usable on a platform such as the Cell. The lessons learned, and parts of the code written, during the implementation of the Cell Transterpreter can be reused in the generation of C from occam-π for the Cell.

5. Conclusions

Our long term goals aim to address the difficulties of programming the Cell processor and other concurrent systems which cannot be parallelised through the use of preprocessor directives and loop level concurrency. We want to establish what a modern, highly concurrent language must provide to be an effective tool for solving modern day computational problems. Using the Transterpreter and occam-π as our starting point, we plan to extend the language to provide direct support for modern hardware. The prototype implementation of the Transterpreter for the Cell has shown that it is possible to use this type of a runtime on an architecture like the Cell and it is a first step in achieving our future goals.

References

[1] Justin Rattner. Multi-core to the masses. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Shekhar Y. Borkar, Pradeep Dubey, Kevin C. Kahn, David J. Kuck, Hans Mulder, Stephen S. Pawlowski, and Justin R. Rattner. Platform 2015: Intel ® processor and platform evolution for the next decade. Technical report, Intel Corporation, 2005.

[3] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[4] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[5] Hans-J. Boehm. Threads cannot be implemented as a library. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 261–268, New York, NY, USA, 2005. ACM Press.

[6] Sony. Playstation III, 2006. http://www.us.playstation.com/PS3.

[7] D. Pham et al. The Design and Implementation of a First-Generation CELL Processor. *Digest of Technical Papers*, pages 184–185, February 2005.

[8] B. Flachs et al. A Streaming Processing Unit for a CELL Processor. Technical report, ISSCC - DIGEST OF TECHNICAL PAPERS, SESSION 7 / MULTIMEDIA PROCESSING / 7.4, 2005.

[9] Christian L. Jacobsen and Matthew C. Jadud. The Transterpreter: A Transputer Interpreter. In Dr. Ian R. East, Prof David Duce, Dr Mark Green, Jeremy M. R. Martin, and Prof. Peter H. Welch, editors, *Communicating Process Architectures 2004*, volume 62 of *Concurrent Systems Engineering Series*, pages 99–106. IOS Press, Amsterdam, September 2004.

[10] Christian L. Jacobsen and Matthew C. Jadud. Towards concrete concurrency: occam-pi on the LEGO mindstorms. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 431–435, New York, NY, USA, 2005. ACM Press.

[11] F.R.M. Barnes and P.H. Welch. Communicating Mobile Processes. In I. East, J. Martin, P. Welch, D. Duce, and M. Green, editors, *Communicating Process Architectures 2004*, volume 62 of *WoTUG-27, Concurrent Systems Engineering, ISSN 1383-7575*, pages 201–218, Amsterdam, The Netherlands, September 2004. IOS Press. ISBN: 1-58603-458-8.

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[13] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes – parts I and II. *Journal of Information and Computation*, 100:1–77, 1992. Available as technical report: ECS-LFCS-89-85/86, University of Edinburgh, UK.

[14] IBM Full-System Simulator for the Cell BE, 2006. http://www.alphaworks.ibm.com/tech/cellsystemsim.

[15] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20(2):85–95, 2000.

[16] A. E. Eichenberger et al. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine achitecture. *IBM SYSTEMS JOURNAL*, 45:59–84, January 2006.

[17] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[18] Mario Schweigler, Fred Barnes, and Peter Welch. Flexible, Transparent and Dynamic occam Networking with KRoC.net. In Jan F Broenink and Gerald H Hilderink, editors, *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering Series*, pages 199–224, Amsterdam, The Netherlands, September 2003. IOS Press.

[19] Damian J. Dimmich and Christan L. Jacobsen. A Foreign Function Interface Generator for occam-pi. In J. Broenink, H. Roebbers, J. Sunter, P. Welch, and D. Wood, editors, *Communicating Process Architectures 2005*, pages 235–248, Amsterdam, The Netherlands, September 2005. IOS Press.

[20] D.C.Wood and J.Moores. User-Defined Data Types and Operators in occam. In B.M.Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering Series*, pages 121–146, Amsterdam, the Netherlands, April 1999. WoTUG, IOS Press.

[21] occam-com mailing list, 2006. http://www.occam-pi.org/list-archives/occam-com/.

[22] Christian L. Jacobsen, Damian J. Dimmich, and Matthew C.Jadud. Native code generation using the transterpreter. In Peter Welch, Jon Kerridge, and Fred Barnes, editors, *Communicating Process Architectures 2006*, Concurrent Systems Engineering Series, Computing Laboratory, University of Kent,Canterbury, CT2 7NZ, England., September 2006. IOS Press, Amsterdam.

[23] GCC Team. Gnu compiler collection homepage. http://gcc.gnu.org/, 2006.