

Mapping and Synchronizing Streaming Applications on Cell Processors

Maik Nijhuis¹, Herbert Bos¹, Henri E. Bal¹, and Cédric Augonnet²

¹ Vrije Universiteit, Amsterdam, The Netherlands
{maik, herbertb, bal}@cs.vu.nl

² INRIA - LaBRI, Université Bordeaux I, France
cedric.augonnet@inria.fr

Abstract. Developing streaming applications on heterogenous multi-processor architectures like the Cell is difficult. Currently, application developers need to know about hardware details to deal with issues like scheduling, memory management and communication/synchronization. Worse, with multiple alternatives for communication available, developers spend significant time picking the most appropriate one. A poor choice often results in bad performance. With **Cell-Space**, we shield users from hardware details without compromising performance. Its runtime is based on an evaluation of the different communication primitives. In **Cell-Space**, developers specify a streaming application as a data flow graph of interacting components. Both task- and data-parallelism are easily expressed and advanced features such as dynamic reconfiguration are fully supported. Beneath a simple interface we include a slew of optimizations not present in other Cell run time environments. We demonstrate the impact of these optimizations and show that **Cell-Space** applications can efficiently exploit the resources offered by the Cell.

1 Introduction

Streaming applications underly a variety of application domains including audio/video, networking, and processing of extremely large data sets. Moreover, much of the consumer electronics industry hinges on streaming. As streaming data is only valid for a limited time, we are forced to process the data in line. On the one hand, keeping up with growing data rates and processing demands is challenging even on modern processors. On the other hand, streaming applications exhibit much potential parallelism. For instance, they may (a) process data in a pipeline of stages, or (b) spread it in a SIMD fashion over a number of identical functional units, or (c) divide a stream in sub-streams (e.g., audio and video) and process each of them differently. In fact, we would probably like to use complex combinations of data and task parallelism.

Heterogeneous multi-core processors like the Cell [1] appear to be a perfect match for these applications. After all, we can run the control part of an application on a Cell's general purpose Power core (the Power Processing element, or PPE) and push all data processing to its specialized SIMD RISC cores (known as Synergistic Processing Elements, or SPEs). The SPEs' 128-bit SIMD organization and fast, private memory make them ideal for stream processing. With eight such cores on a die, the Cell is

one of the most powerful processors currently available. Moreover, vendors often pack multiple Cells in a single blade server. As a result, the Cell is used in machines ranging from game consoles to supercomputers [2].

However, in practice, complex communication and scheduling requirements make streaming applications challenging even on single core architectures. Heterogeneous multi-cores like the Cell add yet another difficulty layer. The question is then: why is it so difficult to map streaming applications on the Cell? In our experience, the problem is caused primarily by the need to find efficient solutions to the following issues:

1. Scheduling and resource utilization (load-balancing);
2. Memory management with distributed memory and inter-core data transfers;
3. Communication and synchronization (efficient notification messages).

Due to lack of high-level programming support, all of these issues have to be handled explicitly by the application programmer, which implies that the programmer has to worry about low-level, architecture-specific details. Making the wrong implementation decision results in poor performance, as all of the above issues are crucial for efficiency. Asking application developers to worry about low-level hardware-specific optimizations borders on the unreasonable and leads to poor portability.

To make matters worse, it is very difficult for the programmer to make an informed decision about *which* mechanism to use for low-level implementation issues like how to handle data transfers and synchronizations. The Cell offers a range of options but it is unclear which are most suitable for what purposes. To achieve good performance, programmers are forced to consider a host of design alternatives. For instance, they must worry about: the pros and cons of interrupts versus DMA, the optimal size of code executing on SPEs, how to split up their applications in components, how to schedule jobs on SPEs and how to decide under what circumstances which parts of the application should be scheduled on what SPEs, etc. These issues are in addition to developing multi-buffering schemes which overlap processing, efficient data transfer, and dealing with more than one Cell processor. The problem gets even uglier when application configurations change at runtime (e.g., a TV that adds or removes picture-in-pictures, PiPs, because the user presses a button).

In our opinion such demands on application developers are undesirable, unreasonable, and unnecessary. Experience in the related field of network processors has shown that it also poses a real threat to the success of the architecture.

Contributions. In this paper, we describe **Cell-Space**, a framework for developing streaming applications on the Cell. In **Cell-Space**, developers use a high-level coordination language for constructing an application, in the form of a data flow graph, out of components in a component library (see Fig. 1). The graph is first translated to an intermediate XML-based language and, after various optimizations, compiled to C. When the application is deployed, a runtime system on the PPE schedules the runnable components in a load-balancing fashion over the available Cell processors. The components in turn use a runtime library to run jobs on the SPEs. **Cell-Space** handles all complex Cell-specific issues, including scheduling, memory management, data transfers, communication and notification.

Our main contributions are:

1. A high-level model which helps building streaming applications from components;
2. A runtime system which schedules components over the available cores in a load-balancing fashion with support for reconfigurability;
3. A runtime library which encapsulates all of the previously mentioned complexity such as multi buffering, synchronization mechanism, code size adaptation, etc.

The runtime library itself is based on an additional contribution, which is that we are the first to report on the relative merits of the various communication and notification mechanisms of the Cell. We evaluated these mechanisms and encapsulated the most optimal mechanisms behind a convenient API. Finally, we evaluate the system not just by means of synthetic micro-benchmarks, but also by way of real applications.

To the best of our knowledge, we are the first to design and implement such an integrated approach for developing for Cell-like architectures which supports reconfigurability. Moreover, even the various parts of **Cell-Space** offer advantages over competing projects. For instance, the Cell runtime library in **Cell-Space** is considerably friendlier than related projects like ALF and Charm++ [3,4]. At the same time, it provides a slew of optimizations not present in its counterparts. Other run time systems exist that schedule data flow graphs and balance the load over available processors, and some, like StreamIt [5], even support the Cell processor. However, none of them seem to have support for reconfigurability.

Paper outline. The focus of this paper will be on the components responsible for ensuring good performance and utilization: the runtime library and, to a lesser extent, the runtime system. Specifically, we will look at the way applications are structured conceptually, how they are scheduled by the runtime, and how they use the runtime library for efficient communication and synchronization. The front-end and intermediate representation are discussed in more detail in [6].

The remainder of this paper is organized as follows. First, we give a high-level overview of **Cell-Space** in Sect. 2. Section 3 presents its implementation as well as the various optimizations we perform. The impact of these optimizations and the overhead of **Cell-Space** is evaluated in Sect. 4. Finally, Sect. 5 presents related work and Sect. 6 concludes the paper.

2 The Cell-Space Architecture

Figure 1a sketches a high-level overview of the **Cell-Space** development model. Developers use a high-level front end for constructing data flow applications from a library of **Cell-Space** components. The model has no implicit preference for any particular front end. Obvious candidates for the front-end are GUI-based environments in which applications are constructed by clicking together components graphically. The only requirement is that the front-end generates programs in the **Cell-Space** intermediate language which presents an application as an XML description of a data flow graph consisting of connections and (possibly nested) components.

The XML-based intermediate language is not just a convenient target for the front-end and an easy-to-parse input for the back-end. It also serves as source and target

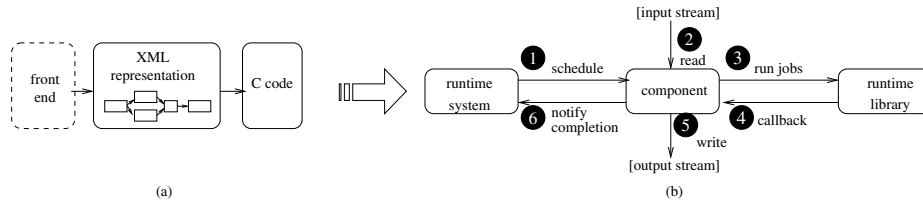


Fig. 1. High-level Cell-Space model

for various XSL transformations and optimizations, such as loop unrolling and conversion of data parallelism to task parallelism where possible. As an aside, in practice the language is quite readable for end users (sufficiently so that thus far we have not yet bothered to write a mature front end, writing all of our applications directly in XML).

After the XML transformations, the intermediate representation is compiled to C and eventually linked to the Space runtime system (*Space-RTS*, see Fig. 1b). *Space-RTS* is responsible for scheduling the components for which all inputs are available on the various processors in the system. The components themselves read data from their input streams. Components may use the Cell runtime library (*Cell-RTL*) for running jobs on the SPEs. *Cell-RTL* notifies the components using a call back mechanism when the jobs have completed. When the component itself completes, it notifies *Space-RTS*.

The glue between *Space-RTS* and *Cell-RTL* is formed by so-called Cell components. Towards the *Cell-Space* framework, Cell components act like normal components. At the PPE, they adhere to the standard component interface, and use the primitives provided by the run time system for streaming and event communication. Internally, they use the *Cell-RTL* library for offloading their computations to the SPEs. The basic interaction in *Cell-Space* is illustrated in Fig. 1. *Space-RTS* schedules a Cell component (①) which reads its input streams (②) using *Space-RTS*'s streaming communication interface, and uses *Cell-RTL* to run jobs (③) on the SPEs to process this data. When jobs complete, *Cell-RTL* invokes a call back function (④) in the component, which then writes the output data (⑤) from the SPEs to its output streams. Finally, the Cell component notifies *Space-RTS* (⑥) that it has finished executing and thereby allows *Space-RTS* to schedule its successors in the data flow graph.

Space-RTS – a runtime system for streaming applications. The *Space-RTS* runtime system abstracts the programmer from difficulties of the parallel architecture, such as load balancing, synchronization, and communication between the main components of the streaming application. *Space-RTS* organizes the components in a data flow graph that corresponds to the graph in the intermediate XML representation and provides both streaming and event communication primitives to the components. Using central and distributed queues, *Space-RTS* dynamically balances the load over the available processors. In addition, it fully supports advanced constructs, such as end-user event handling and dynamic reconfiguration. Due to its modular design, *Space-RTS* can easily be extended to support even more advanced applications in the future.

Reconfigurability is sparked by events that are either caused by user actions (e.g., a button to add a picture-in-picture), or generated by other components. It is a complex

operation, that requires removing, adding, and changing components in a running data flow graph. We handle asynchronous events by buffering them in event queues, which are periodically emptied by a manager component.

Reconfiguration of the data flow graph is supported by the general component interface which supports combining several components in groups. Components can be dynamically created, destroyed, grouped, and connected at run time. To avoid race conditions, the application parts that are reconfigured are made idle before reconfiguring. As we control the activity within the data flow graph, this operation is always possible.

Dynamic load balancing greatly assists dynamic reconfiguration. With static load balancing, the compiler generates a schedule for each possible configuration. The number of schedules grows exponentially with the number of configuration options. Our approach is more elegant as it does not require these schedules. Furthermore, since a schedule captures the full application, the application is fully halted at each reconfiguration. In our approach, the parts that are not reconfigured keep running at full speed.

Cell-RTL – a runtime library for running jobs on SPEs. Cell-RTL is the run time library that facilitates Cell programming by providing a simple interface for using the SPEs. Although Cell-RTL has been developed specifically for the Cell, its simple interface and many optimizations may be applied to similar MPSoC architectures as well. The API of Cell-RTL is based on offloading *jobs* to the SPEs. A job is a self-contained application part that performs some computation at an SPE on input data and produces output data. In streaming applications, these computations consist of kernels and filters, to which Cell-RTL passes the local addresses of the input and output data at the SPE. All jobs are represented by a higher-level **Cell-Space** component in a 1 : n relationship. Thus, Cell-RTL is responsible for low-level synchronization and communication to the jobs on the SPEs, while Space-RTS handles the higher-level synchronization between **Cell-Space** components. Functionality-wise, Cell-RTL relieves the programmer of the following difficult tasks:

- SPE management. Cell-RTL performs all SPE management tasks, including initialization, memory management, scheduling, and exception handling.
- Load balancing. Cell-RTL dynamically assigns jobs to the SPEs, based on their availability. When all SPEs are busy, Cell-RTL internally queues new jobs. When an SPE completes a job, Cell-RTL sends a job from this queue to the SPE.
- Communication. Cell-RTL performs all communication with the SPEs, which includes transferring input and output data between main memory and SPE local memory, sending jobs to the SPEs, and sending notification messages to the PPE.
- Synchronization. Because the SPEs run asynchronously to the PPE, Cell-RTL synchronizes the PPE and the SPEs regularly.

In addition, we will now show that Cell-RTL implements a slew of optimizations like multi-buffering, job chaining, and efficient communication.

3 Implementation

Conceptually, the responsibilities of Cell-RTL and Space-RTS mentioned in the previous section are straightforward. For instance, Space-RTS should track dependencies

and schedule components when all required inputs are available. Similarly, a Cell-RTL job has input and output buffers. Cell-RTL needs to DMA the input data to the SPE's local memory and relay results back to the components running on the PPE. However, efficiently implementing these communication and synchronization mechanisms requires a lot of knowledge about the low-level details of the Cell processor.

For **Cell-Space**, we systematically analyzed and evaluated various alternatives to arrive at a highly efficient runtime. As mentioned earlier, by hiding the details behind Space-RTS and Cell-RTL, **Cell-Space** shields developers from the complex implementation details and tradeoffs. Nevertheless, we believe that both the issues we considered and our findings are essential for anyone developing applications or runtimes for Cell-like processors. For this reason, we now discuss the most important results.

3.1 Asynchronous Notification

The nature of synchronization on processors like the Cell is such that upon completion of a computation cores need to notify other cores. This is typically done by means of a small identifier, such as an integer. In **Cell-Space**, when a job is complete, the SPE sends a single 32 bit notification message to the PPE. The Cell processor has two special-purpose mechanisms which are intended for transferring these messages, namely interrupts and outbound mailboxes. The default DMA communication mechanism can also be used. Figure 2 shows the three approaches, which are described below.

For evaluation purposes, we created three versions of Cell-RTL using interrupts, outbound mailboxes, and DMA, respectively. We describe each of them below. The full evaluation is described in Sect. 4, but as a preliminary result we mention that, surprisingly perhaps, DMA outperforms both special-purpose mechanisms. Cell-RTL therefore uses DMA for notifications by default.

Interrupts. Using an interrupt outbound mailbox, an SPE can trigger an interrupt at the PPE. The PPE then reads the 32 bit mailbox content. The PPE runs a separate thread that is woken up at every SPE interrupt. It is similar to `softirqs` in the Linux kernel [7], as this thread does not run in strict interrupt mode.

Although this approach is relatively straightforward, it has two main disadvantages. First, interrupts are costly because they are handled by the OS [8]. When an interrupt arrives, the interrupt handler is invoked. Also, the OS makes a context switch to the interrupt handling thread. Second, interrupt communication does not scale because there is only one interrupt mailbox for the entire Cell processor, which is shared among all SPEs. When one SPE has sent an interrupt message, the other SPEs have to wait until the PPE has processed it before they can send another interrupt message.

Mailbox. Besides the interrupt outbound mailbox, there is one normal outbound mailbox per SPE, which also holds a 32 bit message. When the SPE uses this mailbox to send a message to the PPE, no interrupt is generated. Therefore the PPE needs to poll the mailbox periodically to see if a new message is available.

This approach has two different disadvantages. First, polling the outbound mailbox for new messages incurs some overhead because it requires a system call to the OS. Moreover, the mailbox resides on the SPE, and not on the PPE. For each poll, the

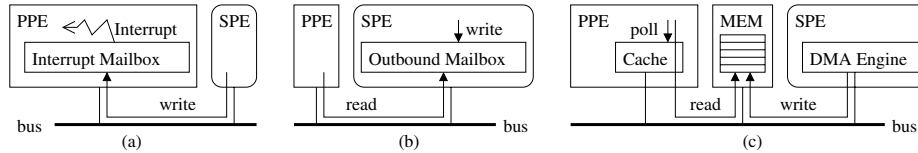


Fig. 2. Notification approaches: (a) Interrupts, (b) Mailbox, (c) DMA

PPE makes a transaction over the internal element interconnect bus (EIB) to access the mailbox at that SPE. Second, each SPE has only one mailbox slot. If an SPE wants to acknowledge multiple jobs, it has to wait until the PPE has read the previous message.

In Cell-RTL, the PPE automatically polls all SPEs whenever the application submits a job. The polling rate is therefore automatically adjusted to the job submission rate. The application can also poll or wait explicitly in case it does not submit new jobs.

DMA. The SPEs can also use DMA for sending 32 bit messages to the PPE. Cell-RTL assigns a special DMA region for each SPE to which it writes job acknowledgment messages. These regions contain a fixed number of message slots, which are used in cyclic order.

With the DMA approach, polling is done by simply reading memory. Similarly to NAPI [9], multiple slots can easily be polled at once, draining all pending notifications in one go. Unsuccessful polls are usually done using the PPE's internal cache, without accessing the internal interconnect bus. By enforcing a maximum number of outstanding jobs per SPE, the PPE ensures that the acknowledgment slot for a certain job is always available. The statically allocated circular buffer is an efficient structure as it provides a lock-free producer-consumer channel and incurs no runtime allocation. Similar constructs are used in various other systems [9,10,11].

Unlike the other approaches, the SPE does not have to wait for the availability of a mailbox slot when acknowledging a job. It only has to schedule an asynchronous DMA request. Only when its DMA engine is fully occupied, it has to wait. However, this condition is unlikely to occur, as the SPE DMA engine can queue up to 16 DMA transfers. When the queue has an available slot, the DMA request is handed over to the DMA engine and the SPE immediately continues processing other jobs.

Similarly to the mailbox approach, Cell-RTL automatically adjusts the polling rate to the job submission rate by polling all notification buffers whenever a job is submitted. Again, the application can poll or wait explicitly when it does not submit new jobs.

3.2 Multi-buffering and Chaining

Cell-RTL allows multiple pending jobs at a single SPE for overlapping communication and computation using multi-buffering. Before executing a job, Cell-RTL initiates asynchronous DMA transfers to fetch the input data for other pending jobs. It also sends the output data using asynchronous DMA. While executing a job, Cell-RTL thus transfers both the input data for subsequent jobs as well as the output data from previous jobs.

Spawning small jobs on the SPEs is expensive as each job incurs notification and transfer overheads. Cell-RTL therefore allows for the combination of several jobs into a *job chain*, which is a list of jobs that is scheduled as one entity. Job chaining reduces overhead as many costs, such as those of synchronization or memory allocation, are incurred once per chain, rather than once per job. Job chaining also allows data transfer and memory re-use between jobs in a chain, which eliminates data transfers to and from main memory when the chain contains producer-consumer jobs.

3.3 Starting Jobs: Data Transfer and Asynchronous Execution

The protocol for data transfer from the Space-RTS streams to Cell-RTL jobs minimizes copying (see Fig. 3). When the component reads or writes a stream, it receives the address in memory of a buffer in the stream from Space-RTS. The component in turn enters this address in the job description of the SPE jobs. Cell-RTL then submits the addresses to the SPEs. The PPE performs no data copying at all, it only copies addresses. The SPE executing the job still transfers the data to its local memory using DMA as it can not directly access main memory.



Fig. 3. Zero-copy protocol

A normal **Cell-Space** component, which only uses the PPE, returns control to **Cell-Space** after it has finished running and all output data streams have been written. When a Cell-RTL component has submitted its jobs to Cell-RTL, it does not wait until Cell-RTL has finished running these jobs. Instead, it returns control to **Cell-Space** which allows **Cell-Space** to run other components, including other Cell-RTL components. Running Cell-RTL components asynchronously has important advantages. First, there is no context switching or thread management overhead as this approach requires only a single thread. Second, the number of active Cell-RTL components is unlimited. Cell-RTL always accepts new jobs and maintains an internal queue of pending jobs at which jobs are put if all SPEs are busy. Third, the main processor and the SPEs are optimally used. The main processor is always available for normal components as Cell-RTL components do not wait. The SPEs are optimally used as Cell-RTL receives all jobs as soon as Space-RTS schedules the corresponding Cell-RTL component.

4 Evaluation

We evaluate **Cell-Space** using two first-generation Cell processors. Each Cell processor has eight available SPEs. As **Cell-Space** distributes load over both processors, we effectively have 16 SPEs and two PPEs. Both the PPEs and the SPEs run at 3.2 GHz. We measure execution time using the built-in decremter register, which ticks every 120 clock cycles. We repeated the experiments 11 times, after which we took the average.

We first measure the overhead of Cell-RTL using a synthetic SPE benchmark function, which executes a fixed number of cycles on the SPE when Cell-RTL invokes it. The total number of cycles spent in all invocations of this function on all SPEs is divided by the number of SPEs, which results in the average effective execution time (AEET).

On the PPE, we measure the total execution time (TET) on the application. The measurement starts just before the application submits the first job to Cell-RTL and

ends when all jobs have finished. It does not include initialization, e.g., set up cost, and finalization, e.g., shutdown cost and printing the result of the measurement.

The relative and absolute overhead of using Cell-RTL are derived from the AEET and the TET. The absolute overhead is the TET minus the AEET, which is zero in the optimal case. The relative overhead is the absolute overhead divided by the AEET, and is given as a percentage. These figures thus include the cost of DMA transfers and the overhead of Cell-RTL. Cell-RTL has overhead both on the PPE side in the interaction with the application and the SPEs, and in the management code that runs on the SPEs.

4.1 SPE Functions

We determine typical computation to communication ratios for SPE jobs by analyzing several SPE functions that are used in real applications. We have created stand alone SPE programs that run the specified function and measure the number of cycles used. The total input and output data size is set to one quarter of the total SPE local memory, which allows multi-buffering when these functions are used in real applications.

All functions in the SPE function library perform image processing. The input and output pixels for these functions are represented using one byte per color component. Using SIMD optimizations, they perform the following operations:

- The maximum function takes the maximum of corresponding values in its input buffers. It processes 64 pixels of one byte at once.
- Yuv2rgb converts color pixel data in YUV format to RGB format. It converts 16 pixels at once.
- IDCT performs an Inverse Discrete Cosine Transform of 8x8 pixel blocks.
- The convolution functions apply a 5x1 or non-separated 3x3 Gaussian blurring kernel to the input image. The input image contains a border of 8 pixels on the left and right sides, which is not present in the output image. With the 3x3 kernel, additional borders of 1 pixel are added at the top and bottom of the input image.
- The geometric transformer applies an affine transformation to the input image. Its input is a block of pixels from the input image. Its output is the corresponding block of pixels in the output image. Its arguments include the transformation matrix, and the position of both the input and output block in the full images.

Table 1 lists the total number of bytes transferred for each kernel along with a measured decremter tick count. From these figures, we compute the number of cycles per transferred byte. This value differs by an order of magnitude. We take this into account when evaluating Cell-RTL because the computation to communication ratio has a significant impact on the overhead of Cell-RTL.

4.2 Notification Approaches

We evaluate the different notification approaches modes described in Sect. 3.1 using a synthetic benchmark application. The benchmark uses the three approaches with 16384 jobs with 32400 input bytes and 32400 output bytes. The number of decremter ticks per job is 540 or 5400 which leads to 1 or 10 clock cycles per transferred byte, respectively. This ratio complies with the results from Sect. 4.1.

Table 1. Computation to communication ratios for various kernels

Kernel	Bytes	Ticks	Cycles/B
Maximum	61952	346	0,670
yuv2rgb	61444	379	0,740
IDCT 8x8	51208	643	1,507
5x1 convolution	66740	4032	7,250
3x3 convolution	64238	4560	8,518
geom. transform	51248	8440	19,763

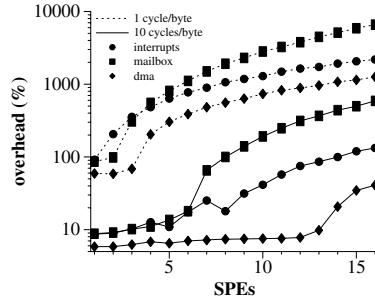


Fig. 4. Notification approach comparison

Figure 4 shows the results of running these benchmarks on 1 to 16 SPEs. Note the logarithmic scale of the y axis. The figure shows three important results:

1. Computationally dense kernels should be preferred as we observed that a high computation to communication ratio implies low overhead. Overhead is greatly reduced by increasing the number of cycles per byte, for example, by combining multiple operations that would otherwise be separate jobs.
2. The overhead increases with the number of SPEs, because there is more resource contention. Also, load imbalance is more likely with more SPEs.
3. There are considerable differences between the various acknowledgment modes. The DMA mode clearly outperforms the interrupt mode, which in turn outperforms the mailbox mode. We conclude that the special interrupt and mailbox communication primitives in the Cell do not provide any added value over the default DMA communication primitive. We have therefore chosen DMA as the notification mechanism within Cell-RTL. In our next experiments, we will only use DMA.

4.3 Multi-buffering

Cell-RTL performs multi-buffering on both input and output data. We evaluate this optimization by varying the maximum number of jobs that Cell-RTL concurrently processes on a single SPE. With one job per SPE, multi buffering is not possible. The multi-buffering opportunities increase with the number jobs per SPE, however, when jobs have 64kB of data, an SPE can only hold the data of three jobs because it has limited memory.

Figure 5 shows that for small jobs with 1 cycle/byte, running multiple jobs per SPE increases overhead. Because the memory bus is overloaded, processing the extra job slots increases overhead. With 10 cycles/byte, multi-buffering decreases overhead from 13 % to 7.5 % with 13 or less SPEs. Beyond 13 SPEs, Cell-RTL's aggressive multi-buffering strategy overloads the bus with DMA transfers and overhead increases again. Fortunately, job chaining overcomes this problem, as explained below. Since we mainly run large jobs, we use 2 jobs per SPE in our other experiments because this setting yields the best results for large jobs.

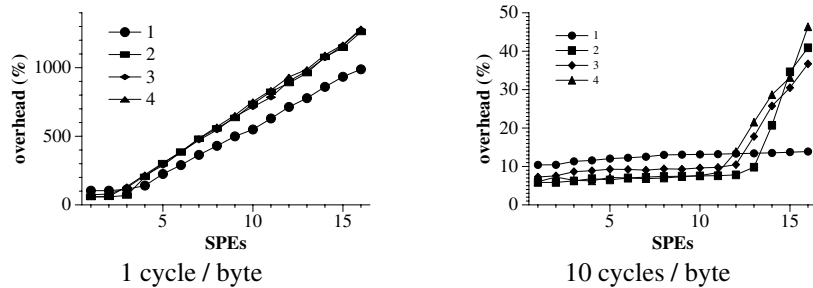


Fig. 5. Evaluation of multi buffering: 1 to 4 pending jobs per SPE

4.4 Job Chaining

For evaluating the impact of using job chaining and persistent data, we run the synthetic benchmark application using three modes:

- Basic. No optimizations are performed.
- Chaining. Instead of scheduling 16384 individual jobs, we schedule 4096 chains with 4 jobs each, or 1024 job chains with 16 jobs each.
- Persistent data. We perform chaining and add persistent input buffers of 12960 or 32400 bytes, which is 20% or 50% of the total data, respectively. We only transfer the persistent buffer with the first job of each chain, however, it remains on the SPE during the execution of the chain. All jobs in the chain can therefore access and even modify this buffer. We reduce the normal input and output buffers by 6480 or 32400 bytes, respectively. The total data size therefore remains equal.

Figure 6 shows the results using 8 and 16 SPEs. On other numbers of SPEs we experienced similar results. Chaining alone effectively reduces overhead: With chaining and 10 cycles/byte, the overhead peak at 16 SPEs (40 %) is completely gone. With 10 cycles/byte, the overhead of Cell-RTL becomes less than 6%. Although using persistent data reduces overhead, the overhead reduction with 20% persistent data is limited because adding an extra persistent buffer increases buffer management overhead. With 50% persistent data, the overhead reduction is more prevalent.

4.5 Application Performance

We now show the effectiveness of *Cell-Space* using several challenging streaming applications. All applications have an output component which normally stores the application output in a file or displays it on the screen. Using these components, we have verified the correctness of the application. Since we are interested in the performance of *Cell-Space*, and the performance of external output devices can be a bottleneck, we replaced the output component by a dummy when performing benchmarks. We examine the following applications:

- The JPiP application decodes 16 motion JPEG (MJPEG) streams with a resolution of 1280x720 and combines them into a 4x4 tiled display. An advanced SPE kernel

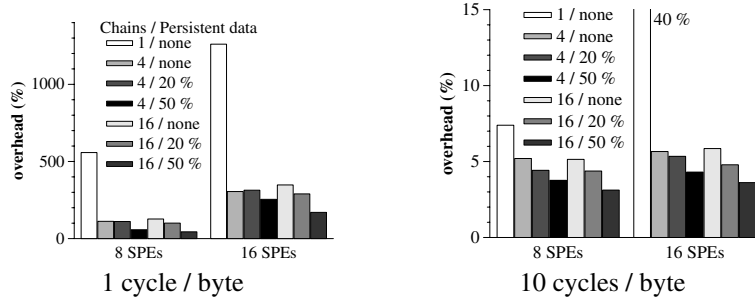


Fig. 6. Evaluation of chaining and persistent data

performs JPEG decompression and down scaling using the IDCT kernel, as mentioned in Sect. 4.1. When all 16 images have been decoded, the PPE blends them into a 4x4 tiled display, which it sends to the output.

- The Edge-Rot application performs edge detection by rotating the input images over 36 different angles. The SPEs perform the rotations using the geometric transformation kernel. For each rotation angle, the PPE generates a border around the input image which is needed for the following convolutions. The SPEs then perform the actual edge detection using four different horizontal first order Gaussian derivative convolution kernels. After each horizontal kernel, the SPEs perform smoothing using four different zero order vertical Gaussian derivative convolution kernels. Then the SPEs take the maximum of all intermediate results and rotate the image with the reverse of the original rotation angle. Finally, the SPEs aggregate the results of all angles by again taking the maximum.
- The Edge-2D application uses a different algorithm for performing edge detection. Instead of rotating the image, it uses rotated two dimensional convolution kernels. For each of 36 different angles, the PPE generates a border around the input image. Similarly to the Edge-Rot application, four different first order Gaussian derivative convolution kernels in the rotated direction are combined with four different zero order Gaussian derivative kernels in the perpendicular direction. The SPEs perform these 16 convolutions and take the maximum of their results. Analogous to the Edge-Rot application, the SPEs aggregate the results of all angles by again taking the maximum.

The Edge-Rot and Edge-2D applications show that **Cell-Space** makes it easy to use different kernels on the SPE: Edge-Rot uses the geometric transformation kernel, the convolution kernel, and the maximum kernel. **Cell-Space** automatically balances the load among the SPEs and performs multi buffering across different kernel types. Using the same kernel across different applications is also easy: Edge-2D uses the same convolution kernel and maximum kernel as Edge-Rot. It only supplies different arguments to the convolution kernel.

Figure 7 shows the speedup of the applications on 1 to 16 SPEs, along with the speedup of the synthetic benchmark application, which runs 1024 chains of 16 jobs without persistent data. With 1 cycle/byte, the speedup of the synthetic benchmark is limited to 5 because of increasing overhead, as shown in Fig. 5. With 10 cycles/byte, linear speedup is achieved as the overhead remains constant.

The speedup of JPiP is limited to 11 because it suffers from load imbalance. It executes only 16 coarse-grained SPE functions in parallel, with varying compute intensities, as they decode different input files. Edge-2D and Edge-Rot achieve speedups of 9 and 12 on 16 SPEs, respectively. They are unable to achieve perfect speedup because of the overhead incurred by running fine-grained functions on the SPEs. Edge-2D performs better than Edge-Rot because its functions are more coarse-grained.

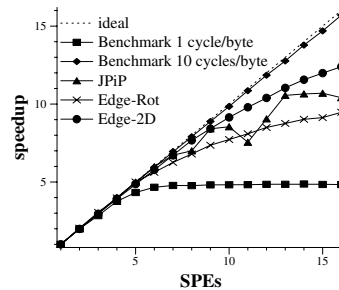


Fig. 7. Application speedup

5 Related Work

Several other projects address the complexity of programming the Cell processor [12]. A challenge faced by most of these systems is that the particularities of Cell-like architectures (heterogeneous cores with local memories) make it hard to apply solutions for scheduling and communication that are based on shared memory [11,13].

Many low-level details of programming the Cell processor are handled by the Linux kernel, which has special system calls for using the SPEs. The `libspe2` library provides a layer on top of the kernel and provides basic functions for using the SPEs [14]. The SPE tool chain provides a small communication library to the code that runs on the SPEs. This environment does not solve problems like load balancing, synchronization between PPE and SPE, and buffer management. These problems have to be solved by adding another layer, such as Cell-RTL.

The Charm++ Offload API [3] and the Accelerated Library Framework (ALF) [4] are similar to Cell-RTL in that they provide an API for offloading jobs to the SPEs in the Cell processor. However, Cell-RTL has many optimizations that are not present in these libraries. Moreover, Cell-RTL is more easy to use due to its simple API. An important difference with ALF is that ALF only supports data parallelism, whereas Cell-RTL supports both task and data parallelism.

Cell SuperScalar [15] and the Single Source Compiler [16] use compiler technology for using the SPEs. The user has to specify the parallel parts of the application, which are then automatically offloaded. The Single Source Compiler focuses on low-level parallelism, such as auto-SIMDization. Cell SuperScalar focuses on high-level parallelism and maintains a data flow graph of pending tasks. Although these approaches are viable for simple applications, we believe they will fail for streaming applications with complex communication patterns and corresponding data dependencies.

In the MultiCore Framework [17], the communication between the PPE and the SPEs resembles streaming. However, full streaming applications with multiple interacting components are not supported. Contrary to **Cell-Space**, The MultiCore Framework uses the SPEs synchronously, which results in low resource utilization.

Gedae [18] is similar to **Cell-Space** as it creates applications from high-level data flow specifications. It maps these specifications onto various hardware configurations, including the Cell. Although the mapping process is dynamic, Gedae statically allocates resources for the application components, whereas **Cell-Space** uses dynamic load balancing. Since Gedae focuses on data processing applications that do not have user interaction, we believe it does not support dynamic reconfiguration.

Several frameworks for developing streaming applications have emerged, of which the StreamIt language is the most notable example [19]. StreamIt expresses streaming applications using sequential pipeline, parallel split/join, and feed back loop primitives, which are also supported by **Cell-Space**. The StreamIt compiler detects parallelism in the application and maps it onto an homogeneous MPSoC [20]. This mapping is static, whereas **Cell-Space** uses dynamic load balancing. The StreamIt compiler supports the Cell architecture using the Multicore Streaming Layer (MSL) [5], which executes application kernels on the SPEs using static or dynamic load balancing. Contrary to **Cell-Space**, the main processor only runs control code and can not be used for computations in this system. Also, StreamIt does not support dynamic reconfiguration and event communication, whereas **Cell-Space** fully supports these advanced features.

6 Conclusions

This paper presents **Cell-Space**, a framework for developing streaming applications for heterogeneous multi-cores like the Cell. Developers construct applications by means of data flow components that are then scheduled on the Cell's Power core by a runtime system which offers a convenient streaming communication interface. Components may use the Cell runtime library for running jobs on the processor's synergistic processing elements. The framework shields developers from all low-level hardware mechanisms that are essential for performance. By carefully evaluating the relative merits of the individual mechanisms and encapsulating these under a simple API, **Cell-Space** greatly simplifies the development of streaming applications on the Cell without compromising performance. We have evaluated performance by means of real applications such as a 16-stream tiled display, and two edge detection algorithms.

Acknowledgments. We would like to thank Frank Seinstra for supporting the application development. This work is supported by the Dutch government's STW/Progress project 06397.

References

1. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the Cell multiprocessor. *IBM Journal of Research and Development* 49(4/5), 589 (2005)
2. Williams, S., Shalf, J., Oliner, L., Kamil, S., Husbands, P., Yelick, K.: The potential of the Cell processor for scientific computing. In: *Proc. 3rd conf. on Computing Frontiers*, pp. 9–20. ACM Press, New York (2006)

3. Kunzman, D., Zheng, G., Bohm, E., Kalé, L.V.: Charm++, offload API, and the cell processor. In: Proc. Workshop on Programming Models for Ubiquitous Parallelism, Seattle, WA, USA (September 2006)
4. IBM: Accelerated Library Framework Programmer's Guide and API Reference (March 2007)
5. Zhang, X.D., Li, Q.J., Rabbah, R., Amarasinghe, S.: A lightweight streaming layer for multicore execution. In: Workshop on Design, Architecture and Simulation of Chip Multi-Processors, Chicago, IL (December 2007)
6. Nijhuis, M., Bos, H., Bal, H.E.: A component-based coordination language for efficient reconfigurable streaming applications. In: Proc. Intl. Conf. on Parallel Processing, Xi'An, China (September 2007)
7. Bovet, D.P., Cesati, M.: Understanding the Linux Kernel, 3rd edn. O'Reilly, Sebastopol (2005)
8. Welsh, M., Basu, A., von Eicken, T.: Incorporating memory management into user-level network interfaces. In: Proceedings of Hot Interconnects V (August 1997)
9. Salim, J.H., Olsson, R., Kuznetsov, A.: Beyond softnet. In: Proc. 5th Annual Linux Showcase & Conference, November 2001, pp. 165–172. USENIX Association, Berkeley (2001)
10. Bos, H., de Bruijn, W., Cristea, M., Nguyen, T., Portokalidis, G.: FFPF: fairly fast packet filters. In: Proc. 6th Symposium on Operating Systems Design and Implementation (December 2004)
11. Govindan, R., Anderson, D.P.: Scheduling and ipc mechanisms for continuous media. In: SOSP, ACM SIGOPS, pp. 68–80 (1991)
12. Buttari, A., Luszczek, P., Kurzak, J., Dongarra, J., Bosilca, G.: Scop3: A rough guide to scientific computing on the playstation 3. version 0.1. Technical Report UT-CS-07-595, ICL, University of Tennessee, Knoxville (April 2007)
13. Pai, V.S., Druschel, P., Zwaenepoel, W.: Io-lite: a unified i/o buffering and caching system. ACM Transactions on Computer Systems 18(1), 37–66 (2000)
14. IBM: SPE Runtime Management Library, Version 2.2 (October 2007)
15. Bellens, P., Pérez, J.M., Badia, R.M., Labarta, J.: CellSs: a programming model for the Cell BE architecture. In: Proc. 2006 ACM/IEEE Supercomputing conf., p. 86. ACM Press, New York (2006)
16. Eichenberger, A.E., O'Brien, J.K., O'Brien, K.M., Wu, P., Chen, T., Oden, P.H., Prener, D.A., Shepherd, J.C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., Gschwind, M.K., Archambault, R., Gao, Y., Koo, R.: Using advanced compiler technology to exploit the performance of the cell broadband engine™ architecture. IBM System Journal 45(1), 59–84 (2006)
17. Bouzas, B., Cooper, R., Greene, J., Pepe, M., Prella, M.J.: Multicore framework: An API for programming heterogeneous multicore processors. In: First Workshop on Software Tools for Multi-Core Systems, Manhattan, New York, NY (March 2006)
18. Inc, G., <http://www.gedae.com>
19. Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt: A language for streaming applications. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 179–196. Springer, Heidelberg (2002)
20. Gordon, M., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, San Jose, CA (October 2006)